Software-Projekt I

Prof. Dr. Rainer Koschke

Arbeitsgruppe Softwaretechnik
Fachbereich Mathematik und Informatik
Universität Bremen

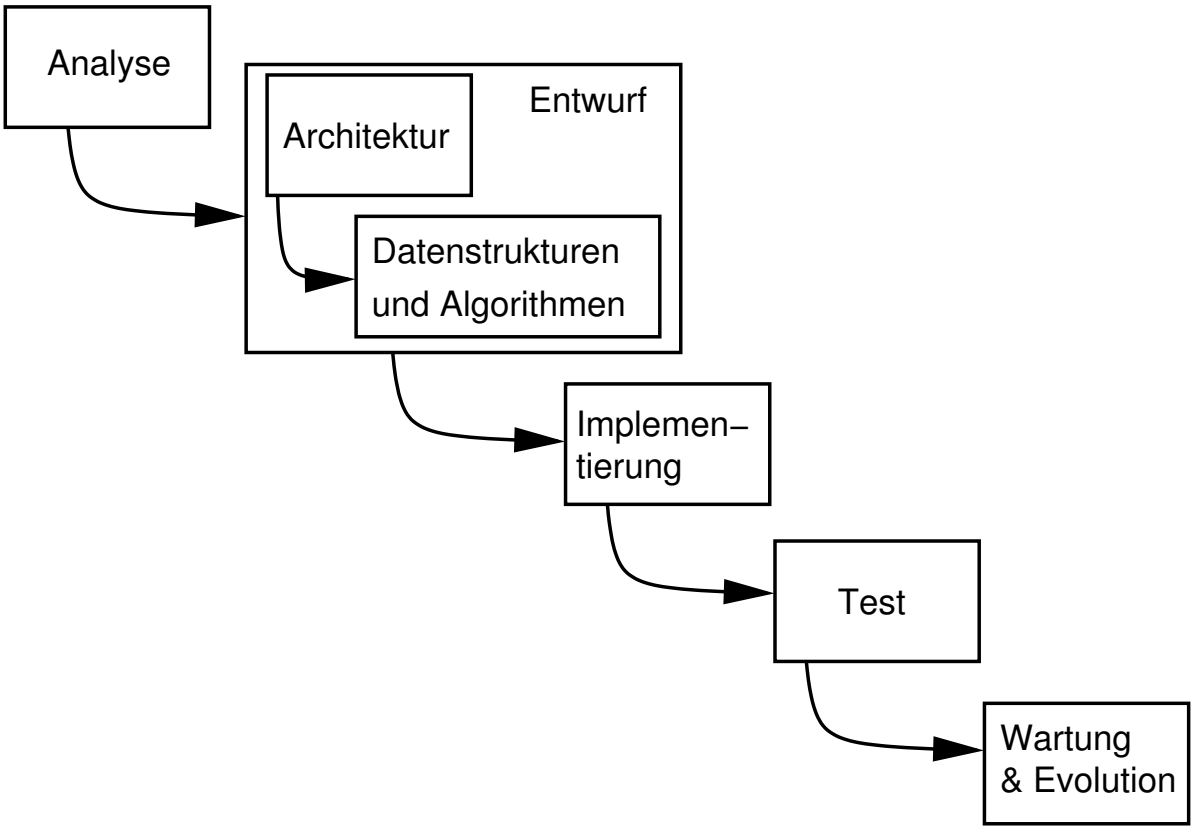
Sommersemester 2013

Software-Architektur I

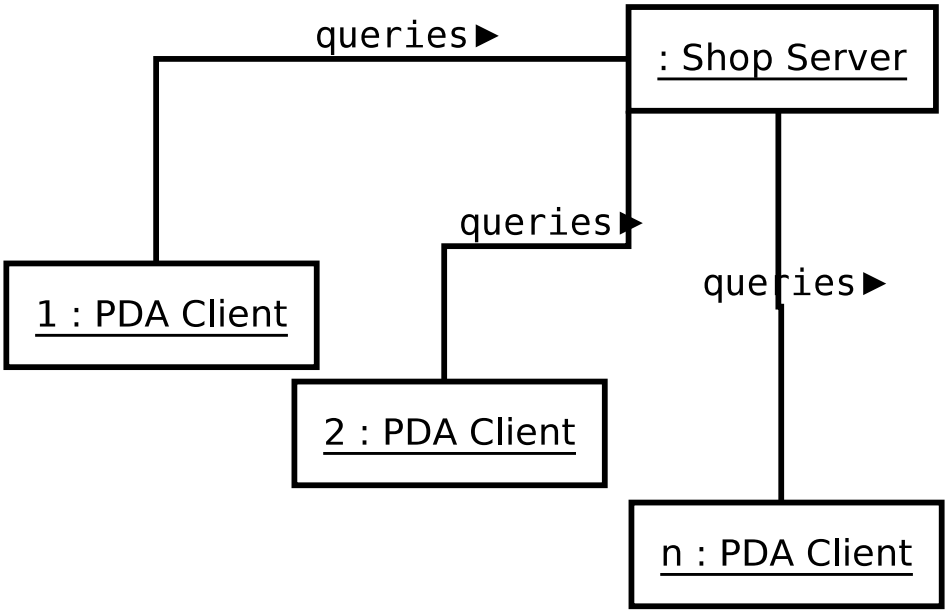
Fragen

- Was ist Software-Architektur?
- Was sind Architektursichten und welche gibt es?
- Was sind die angestrebten Qualitäten einer Architektur?
- Wie entwirft man eine angemessene Software-Architektur?
- Wie kann man eine Software-Architektur bewerten?

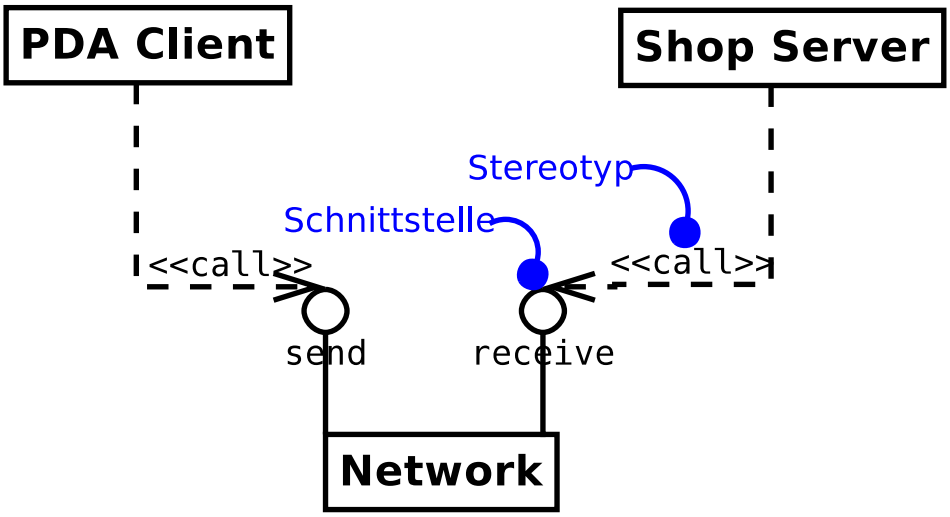
Kontext



Client-Server-Architektur: Dynamische Sicht



Client-Server-Architektur: Statische Sicht



Was ist Architektur?

> ***Architecture*** *is the human organization of empty space using raw material.*
>
> Richard Hooker, 1996.

Definition
Software-Architektur ist die grundlegende Organisation eines Systems verkörpert (IEEE P1471 2002)

- in seinen Komponenten,
- deren Beziehungen untereinander und zu der Umgebung
- und die Prinzipien, die den Entwurf und die Evolution leiten.

Bedeutung von Software-Architektur

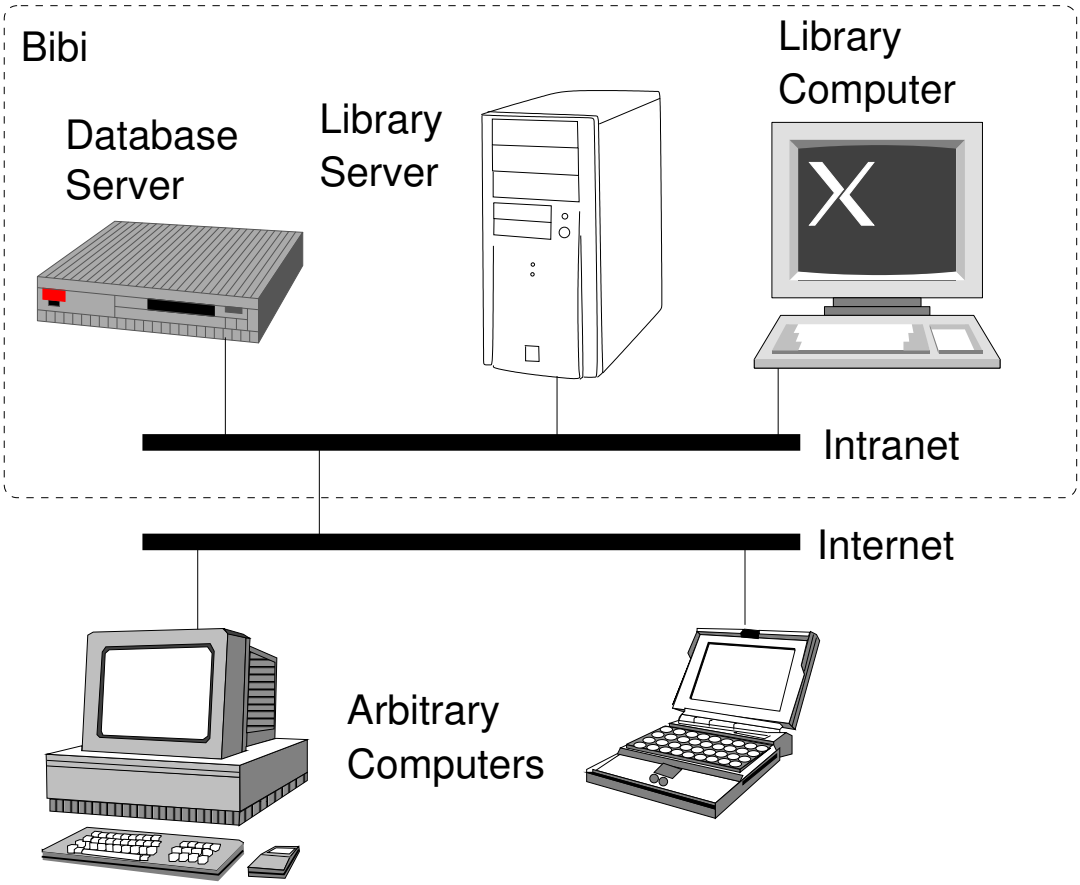
- Kommunikation zwischen allen Interessenten
  - Aufteilung der Arbeit in unabhängig bearbeitbare Teile
  - hoher Abstraktionsgrad, der von vielen verstanden werden kann
- Frühe Entwurfsentscheidungen
  - → nachhaltige Auswirkungen
  - → frühzeitige Analyse
- Transferierbare Abstraktion des Systems
  - → eigenständig wiederverwendbar
  - → unterstützt Wiederverwendung im großen Stil (Software-Produktlinien)

Die Softwarearchitektur (SA) repräsentiert eine hohe Abstraktion eines Systems, die von den meisten Interessenten verstanden werden kann und damit eine Grundlage zum gegenseitigen Verständnis, zur Konsensbildung und zur Kommunikation darstellt.

Die Softwarearchitektur ist die Manifestation früher Entwurfsentscheidungen. Diese frühe Fixierung kann nachhaltige Auswirkungen haben auf die nachfolgende Entwicklung, Auslieferung sowie Wartung und Evolution. SA ist auch die früheste Systembeschreibung, die analysiert werden kann.

Die Softwarearchitektur konstitutiert ein relativ kleines intellektuell fassbares Modell darüber, wie das System struktiert ist und wie seine Komponenten zusammenwirken. Dieses Modell ist eigenständig nutzbar und kann über das spezifische System hinaus transferiert werden - insbesondere kann es für Systeme mit ähnlichen Eigenschaften und Anforderungen wiederverwendet werden, um so Wiederverwendung im großen Stil zu unterstützen (Stichwort: Software-Produktlinien).

Beispielsystem Bibi



Einflussfaktoren/Anliegen (Concerns)

Einflussfaktoren

- Produktfunktionen und -attribute
  - z.B. Performanz, Ansprüche an Zuverlässigkeit, Umfang und Stabilität der Anforderungen
- Organisation
  - z.B. Budget, verfügbare Zeit, Team-Größe und -erfahrung
- Technologie
  - z.B. verfügbare Hard- und Software

Keiner der Faktoren kann isoliert behandelt werden $\rightarrow$ globale Analyse.

Einflussfaktoren

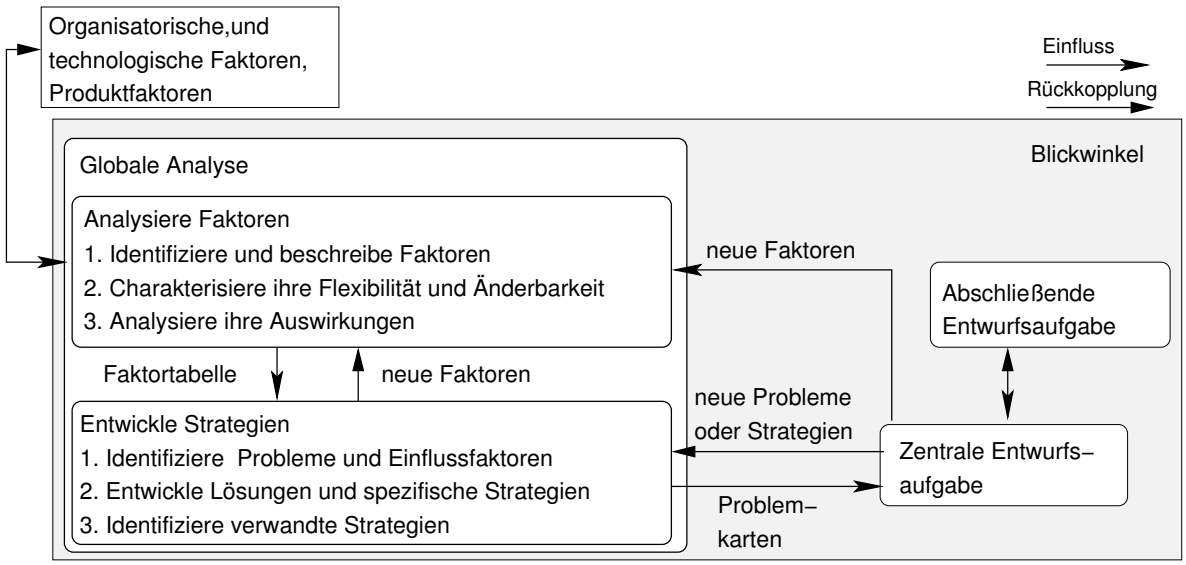
Anmerkung

Ein Faktor ist nur dann relevant, wenn er Einfluss auf die Architektur hat. Test:

- Architektur $A1$ ergibt sich mit Betrachtung des Faktors
- Architektur $A2$ ergibt sich ohne Betrachtung des Faktors

Nur wenn $A1$ und $A2$ sich unterscheiden, ist der Faktor relevant.

Globale Analyse nach Hofmeister u. a. (2000)



Globale Analyse: Analysiere Faktoren

1. Identifiziere und beschreibe Faktoren
   - Faktoren, die viele Komponenten betreffen
   - Faktoren, die sich über die Zeit ändern
   - Faktoren, die schwer zu erfüllen sind
   - Faktoren, mit denen man wenig Erfahrung hat

Beispiele:

- *Umfang funktionaler Anforderungen*
- *Abgabetermin*
- *Stabilität der Hardware-Plattform*

Globale Analyse: Analysiere Faktoren

2.a Charakterisiere ihre Flexibilität

Flexibilität

Kann zu Beginn mit Beteiligten (Manager, Marketingleute, Kunde, Benutzer etc.) über Faktoren verhandelt werden?

Relevante Fragen zur Flexibilität (am Beispiel *Auslieferung der Funktionalität*):

- Kann der Faktor beeinflusst oder geändert werden, so dass der Architekturentwurf vereinfacht werden kann?
  - *Auslieferung der Funktionalität ist verhandelbar. Weniger wichtige Funktionalität kann in späterem Release ausgeliefert werden.*
- Wie kann man ihn beeinflussen?
  - *Nur nach rechtzeitiger Absprache mit dem Kunden.*
- Zu welchem Grad kann man ihn beeinflussen?
  - *Der Kunde hat Mindestanforderungen, die erfüllt werden müssen.*

Globale Analyse: Analysiere Faktoren

2.b Charakterisiere ihre Veränderlichkeit

Veränderlichkeit

Kann sich der Faktor während der Entwicklung ändern?

Relevante Fragen (am Beispiel *Auslieferung der Funktionalität*):

- In welcher Weise kann sich der Faktor ändern?
  - *Prioritäten könnten sich ändern.*
  - *Anforderungen könnten wegfallen, weil nicht mehr relevant.*
  - *Anforderungen könnten hinzukommen, weil sich Rahmenbedingungen geändert haben.*
- Wie wahrscheinlich ist die Änderung während und nach der Entwicklung?
  - *Moderat während der Entwicklung; sehr hoch danach.*
- Wie oft wird er sich ändern?
  - *Alle 3 Monate.*
- Wird der Faktor betroffen von Änderungen anderer Faktoren?
  - *Änderung technologischer Aspekte (Software-/Hardware-Plattform).*

Globale Analyse: Analysiere Faktoren

3. Analysiere die Auswirkungen der Änderung von Faktoren auf Architektur

Auswirkungen

Was wird von dem Faktor *wie* beeinflusst?

Auswirkungen auf:

- Andere Faktoren
  - *Z.B. moderater Einfluss auf Einhaltung des Zeitplans und damit darauf, was zu welchem Zeitpunkt ausgeliefert werden kann*
- Systemkomponenten
  - *Z.B. geplante Komponenten müssen überarbeitet werden; Komponenten können hinzu kommen oder wegfallen.*
- Operationsmodi des Systems
- Andere Entwurfsentscheidungen
- ...

Faktortabelle zu organisatorischen Aspekten

Faktor	Flexibilität und Veränderlichkeit	Auswirkung
O1 : Entwicklungsplan		
O1.1 : Time-To-Market		
Auslieferung Febr. 2011	Keine Flexibilität	Nicht alle Funktionen können implementiert werden
O1.2 : Auslieferung von Produktfunktionen		
priorisierte Funktionen	Funktionen sind verhandelbar	Die Reihenfolge bei der Implementierung der Funktionen kann sich ändern
O2 : Entwicklungsbudget		
O2.1 : Anzahl Entwickler		
6 Entwickler	Keine neuen Entwickler können eingestellt werden. Entwickler können (temporär) ausfallen.	Moderater Einfluss auf Zeitplan; partielle Implementierung droht

Weitere organisatorische Beispielfaktoren I

O1: Management

- Selbst herstellen versus kaufen
- Zeitplan versus Funktionsumfang
- Geschäftsziele

O2: Personal: Fähigkeiten, Interessen, Stärken, Schwächen mit ...

- Anwendungsdomäne
- Softwareentwurf
- speziellen Implementierungstechniken
- speziellen Analysetechniken

Weitere organisatorische Beispielfaktoren II

O3: Prozesse und Werkzeuge für die Entwicklungsschritte, d.h.:

- Entwicklung (IDE), z.B. *Eclipse*
- Konfigurationsmanagement, z.B. *Subversion*
- Systembau (Build) z.B. *Maven*
- Test, z.B. *JUnit*
- Release, z.B. *Installer*

O4: Entwicklungszeitplan

- Time-To-Market
- Auslieferung der Produktfunktionen
- Release-Zeitplan

Weitere organisatorische Beispielfaktoren III

O5: Entwicklungsbudget

- Anzahl Entwickler
- Kosten von Entwicklungswerkzeugen

Faktortabelle zu technischen Aspekten

Faktor	Flexibilität und Veränderlichkeit	Auswirkung
T2 : Software		
T2.1 : Betriebssystem (BS)		
Bibi-Clients sollen auf verschiedenen BS laufen.	Neuere Versionen alle 3 Jahre	Großer Einfluss auf betriebssyste-mabhängige Komponenten
T2.2 : Benutzerschnittstelle		
GUI soll sich im BS-spezifischen Look&Feel darstellen	Neuere Versionen alle 2 Jahre	Großer Einfluss auf GUI-Komponenten

Technische Beispielfaktoren I

T1: Hardware

- Prozessoren
- Netzwerk
- Hauptspeicher
- Plattenspeicher
- domänenspezifische Hardware

T2: Software

- Betriebssystem
- Benutzerschnittstelle
- Software-Komponenten
- Implementierungssprache
- Entwurfsmuster
- Rahmenwerke

T3: Architekturtechnologie

Technische Beispielfaktoren II

- Architekturstile/-muster und -rahmenwerke
- Domänenspezifische Architekturen oder Referenzarchitekturen
- Architekturbeschreibungssprachen
- Software-Produktlinien-Technologien
- Werkzeuge zur Analyse und Validierung von Architekturen

T4: Standards

- Schnittstelle zum Betriebssystem (z.B. Posix)
- Datenbanken (z.B. SQL)
- Datenformate
- Kommunikation (z.B. TCP/IP)
- Kodierrichtlinien

Faktortabelle zu Produktfaktoren

Faktor	Flexibilität und Veränderlichkeit	Auswirkung
P4 : Verlässlichkeit		
P4.3 : Sicherheit		
hohe Anforderungen an Datensicherheit	nicht flexibel	keine große Auswirkung, wenn Datensicherheit weniger wichtig wird (Architektur könnte vereinfacht werden)

Beispielproduktfaktoren I

P1: Produktfunktionen

P2: Benutzerschnittstelle

- Interaktionsmodell
- Funktionen der Benutzerschnittstelle

P3: Performanz

- Ausführungszeiten
- Speicherbedarf
- Dauer des Systemstarts und -endes
- Wiederherstellungszeit nach Fehlern

Beispielproduktfaktoren II

P4: Verlässlichkeit

- Verfügbarkeit
- Zuverlässigkeit
- Sicherheit

P5: Fehlererkennung, -bericht, -behandlung

- Fehlerklassifikation
- Fehlerprotokollierung
- Diagnostik
- Wiederherstellung

Beispielproduktfaktoren III

P6: Service

- Service-Dienste (Konfigurations- und Wartungsdienste) für den Betrieb des Systems
- Installation und Aktualisierung
- Test der Software
- Wartung und Erweiterung der Systemimplementierung

P7: Produktkosten

- Hardwarebudget
- Softwarelizenzbudget (für verwendete Software)

Globale Analyse: Entwickle Strategien I

1. Identifiziere Probleme und deren Einflussfaktoren

Analysiere Faktorentabelle:

- Grenzen oder Einschränkungen durch Faktoren
  - *Unverrückbarer Abgabetermin erlaubt keinen vollen Funktionsumfang*
- Notwendigkeit, Auswirkung eines Faktors zu begrenzen
  - *Entwurf muss Portierbarkeit vorsehen*
- Schwierigkeit, einen Produktfaktor zu erfüllen
  - *Speicher- und Prozessorbegrenzung erlaubt keine beliebig komplexen Algorithmen*
- Notwendigkeit einer allgemeinen Lösung zu globalen Anforderungen wie Fehlerbehandlung und Wiederaufsetzen nach Fehlern

Globale Analyse: Entwickle Strategien II

2. Entwickle Lösungen und spezifische Strategien

. . . für die Behandlung der Probleme, die sich implementieren lassen und die notwendige Änderbarkeit unterstützen.

Strategie muss konsistent sein zu:

- Einflussfaktor,
- dessen Veränderlichkeit
- und dessen Interaktion mit anderen Faktoren.

Globale Analyse: Entwickle Strategien III

2. Entwickle Lösungen und spezifische Strategien

Ziele:

- Reduzierung oder Kapselung des Faktoreinflusses
- Reduzierung der Auswirkung einer Änderung des Faktors auf den Entwurf und andere Faktoren
- Reduzierung oder Kapselung notwendiger Bereiche von Expertenwissen oder -fähigkeiten
- Reduzierung der Gesamtentwicklungsdauer und -kosten

3. Identifiziere verwandte Strategien

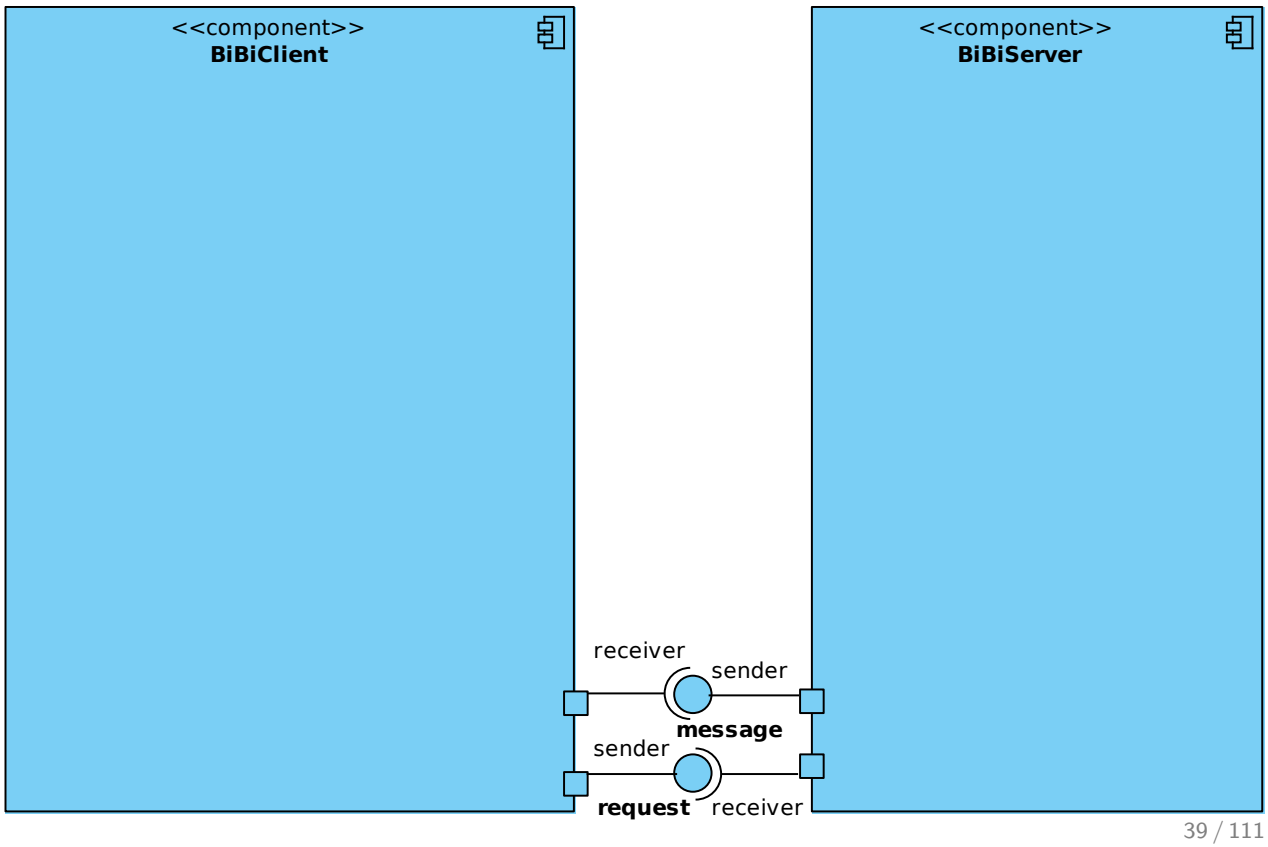
Globale Analyse: Entwickle Strategien IV

Problemkarten (Issue Cards) beschreiben Problem und passende Strategien einmalig:

Name des Problems
Beschreibung des Problems **Einflussfaktoren** *Liste aller Einflussfaktoren*
Lösung *Diskussion einer allgemeinen Lösung* **Strategie: *Name der Strategie A*** *Erläuterung der Strategie A* **Strategie: *Name der Strategie B*** *Erläuterung der Strategie B* . . .
Verwandte Strategien *Referenzen zu verwandten Strategien.* *Diskussion, in welcher Weise sie verwandt sind.*

Strategie

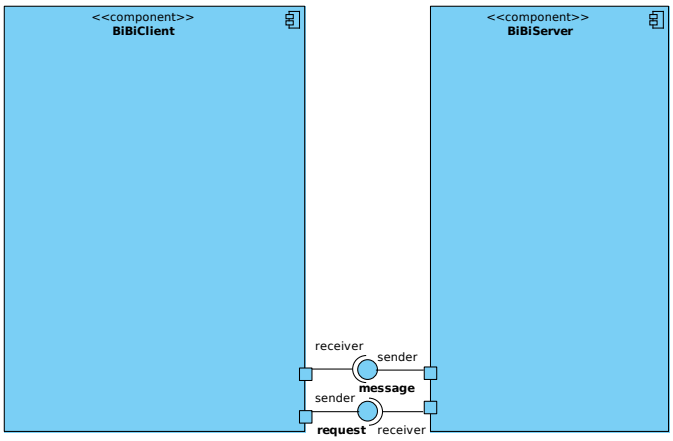
Lege externe Schnittstellen fest.



Wir illustrieren nun an unserem laufenden Beispiel, wie die Hofmeister-Methode eine konzeptionelle Sicht entwirft. Ausgangspunkt sind die Strategien, die wir in der globalen Analyse aufgestellt haben, um mit Entwurfskonflikten umzugehen. Sie werden nun eingesetzt.

Wir beschreiben den Entwurfsprozess top-down, das heißt, wir beginnen im Folgenden mit dem System als eine große Komponente und verfeinern sie weiter. Das ist ein gängiges Vorgehen. Sollen aber existierende Komponenten wiederverwendet werden, dann kombiniert man den Top-Down-Ansatz mit einem Bottom-Up-Vorgehen. Letzterer geht von den existierenden Komponenten aus und baut sie zu einem größeren Ganzen sukzessive zusammen. In der Kombination entwirft man sowohl top-down als auch bottom-up und versucht, die beiden Entwurfsrichtungen aufeinander zuzuführen. Beim reinen Top-Down-Vorgehen kann es dazu kommen, dass existierende Komponenten nicht wiederverwendet werden, weil sie nicht in das Gefüge passen - außerdem können redundante Komponenten entstehen. Erfahrene Architekten beherrschen beide Richtungen.

Wir beginnen hier also zunächst mit dem System als eine einzige Komponente. In der Anforderungsanalyse haben wir uns bereits damit auseinandergesetzt, welche Schnittstellen nach außen das System bieten muss. Diese werden nun eingezogen. Es ergibt sich das folgende Bild mit einer Komponente für den Client und eine für den Server:

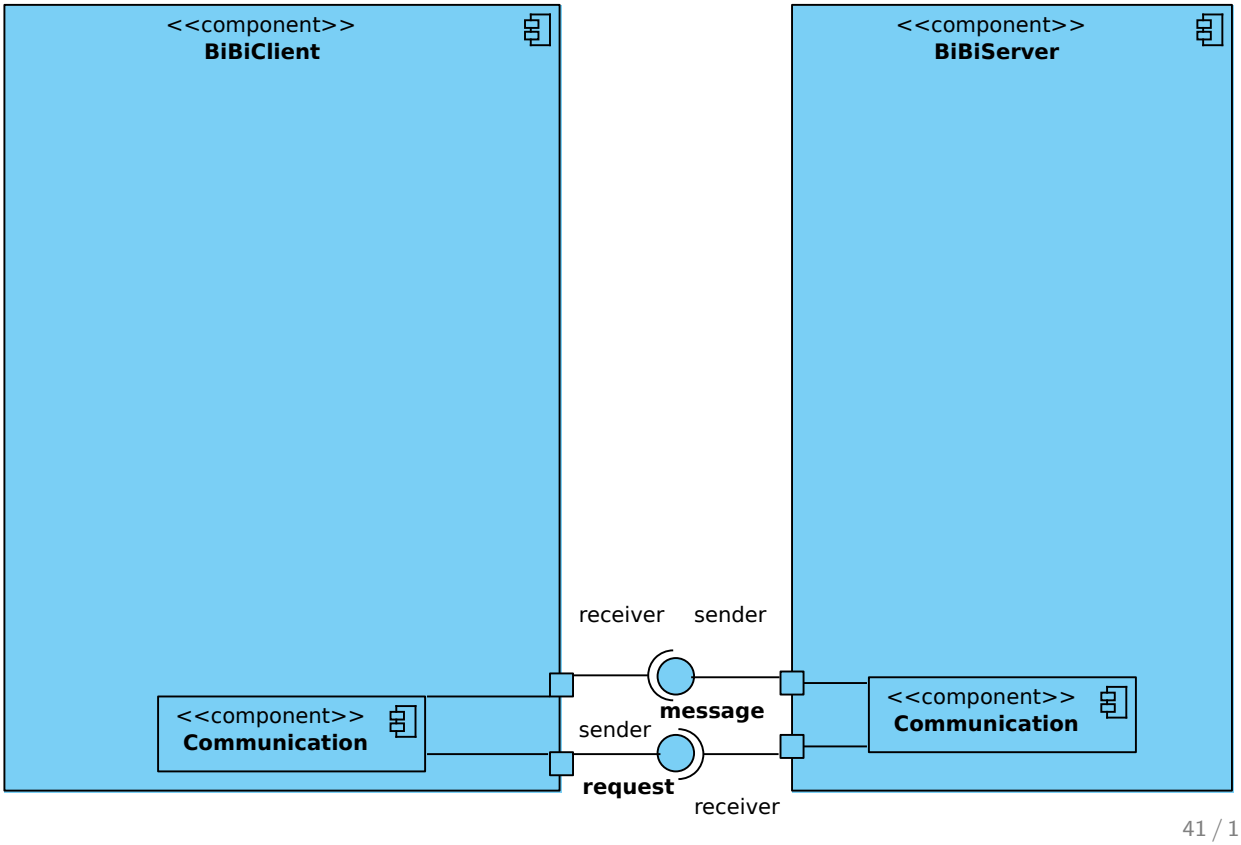


Die Kästchen am Rande der Komponenten sind ihre Ports, das heißt, ihre Schnittstellen nach außen.

Netzwerkanpassung
Weiterentwicklung der Netzwerke und Wechsel auf neue Technologie machen Anpassungen an kommunikationsspezifischen Komponenten notwendig. Für die Wartung steht nur ein Entwickler zur Verfügung. Die Anpassungen müssen schnell vorgenommen werden. **Einflussfaktoren** *O1.1: Time-To-Market* *O2.1: Anzahl Entwickler* *T1.2: Betriebssystem*
Lösung **Strategie: *Kapselung der kommunikationsabhängigen Anteile*** Kommunikationsabhängige Anteile werden in Komponenten isoliert. Diese bilden eine virtuelle Maschine für alle anderen Komponenten.

Strategie

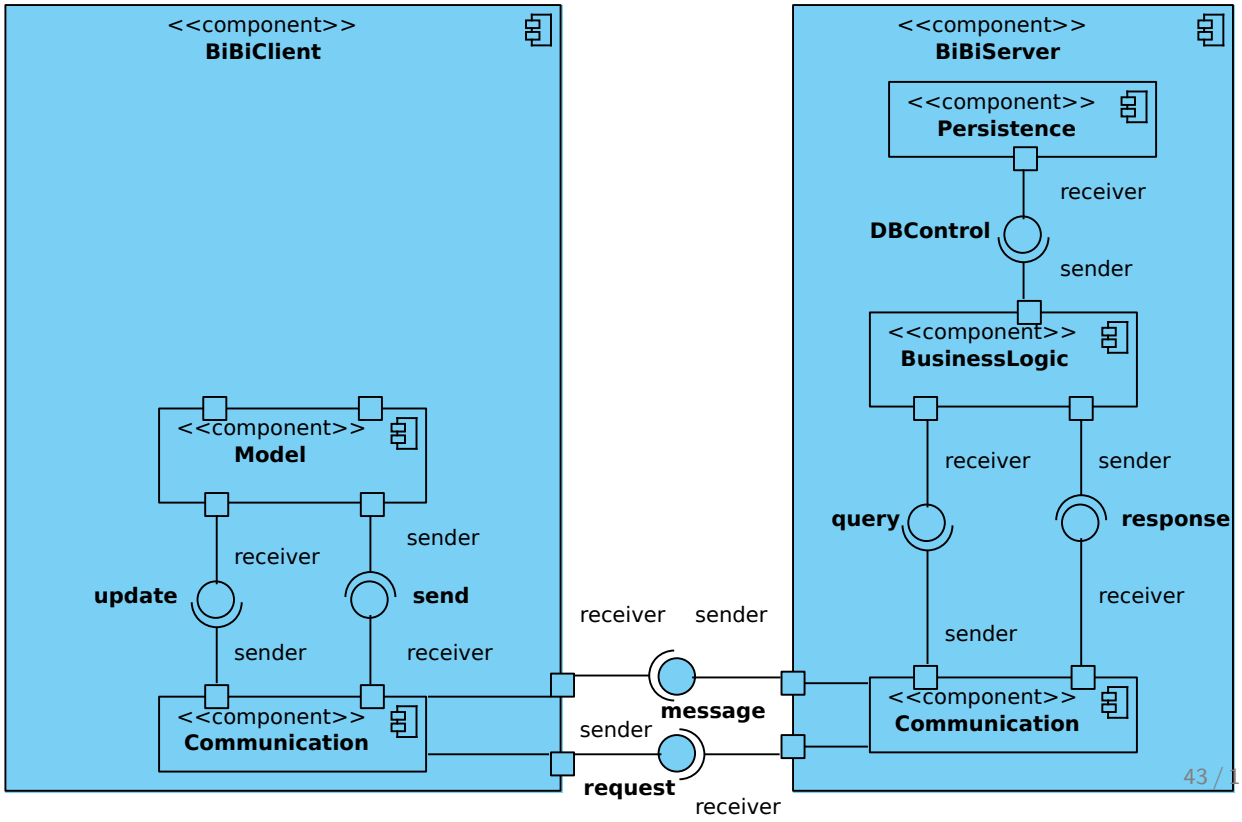
Kapselung der kommunikationsabhängigen Anteile



Datensicherheit
Zugriff auf die Daten muss kontrolliert werden. Daten müssen gegen Verlust gesichert werden. Selbst implementierte Lösungen sind riskant (mangelnde Entwicklererfahrung, nicht genug Budget). **Einflussfaktoren** *P4.3: Datensicherheit.* *O1.1: Time-To-Market* *O2.1: Anzahl Entwickler*
Lösung **Strategie: *Kapselung der Datenhaltung:*** Datenhaltung ist eigene Komponente. **Strategie: *Wiederverwendung*** Einbindung externer wiederverwendbarer Komponenten. **Strategie: *Benutzung eines Datenbankmanagementsystems (DBMS):*** Verwende relationales DBMS. **Strategie: *Zugriff auf Daten nur über Server:*** DBMS ist lokal für Server, nicht sichtbar für Clients.

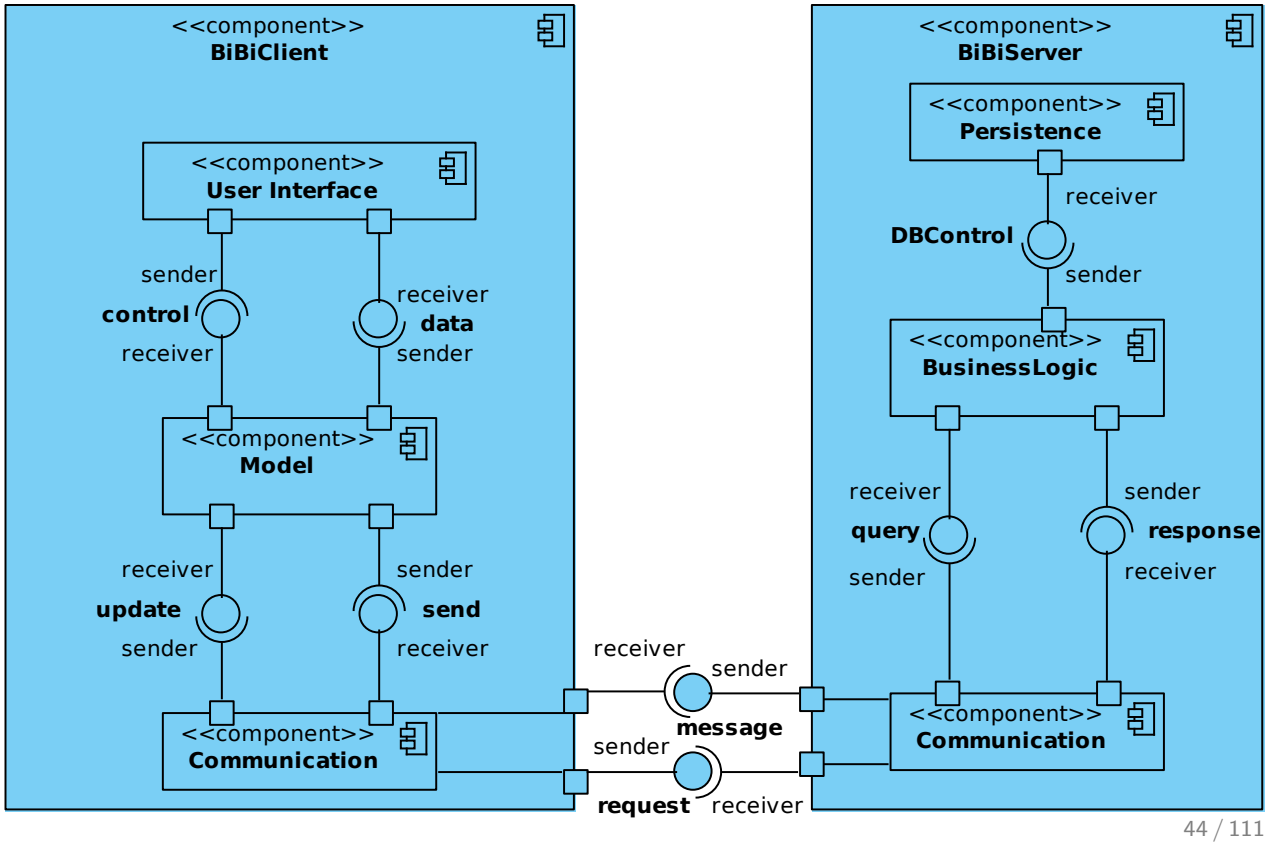
Strategien

Kapselung der Datenhaltung, Wiederverwendung, Benutzung eines DBMS, Zugriff auf Daten nur über Server



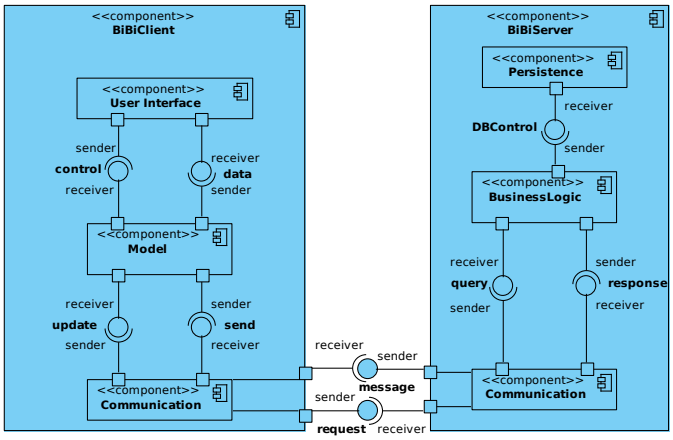
Strategie

Trennung von Logik und Darstellung



Nun führen wir schrittweise Komponenten für Client und Server ein. Diese Aufteilung folgt den vorher identifizierten Strategien. Die inneren Komponenten werden über innere Konnektoren verbunden. Hier gilt die Regel, dass eine Verbindung zwischen einem Port und einer Rolle nur möglich ist, wenn die zugehörigen Komponenten und Konnektoren im selben anderen Element (Komponente oder Konnektor) enthalten sind. Mit anderen Worten, die Verbindung kann keine Abstraktionsebene durchqueren. Näheres zu Komponenten, Konnektoren, Ports und Rollen folgt später.

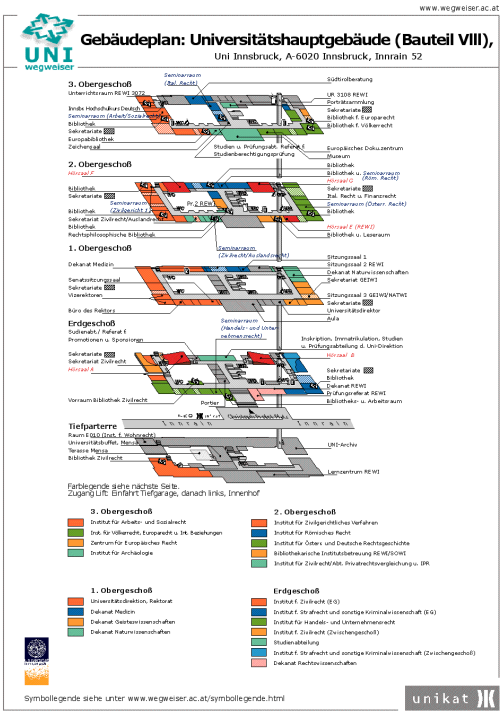
Wir stellen fest, dass die Komponenten und Konnektoren wie gefordert Teil der selben Komponente sind. Damit ist das Diagramm in diesem Punkt wohlgeformt.

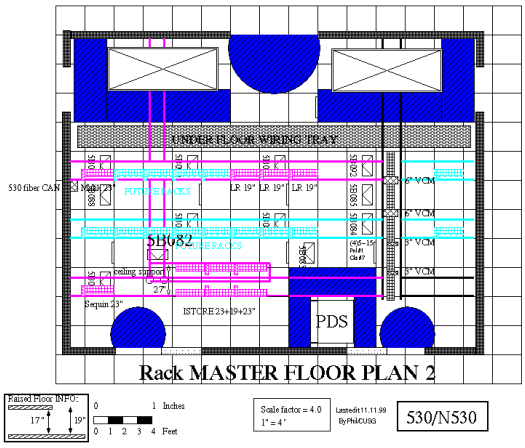


Beispiele

Ambitionierter Zeitplan
Abgabetermin ist fix, Ressourcen sind begrenzt. Möglicherweise können nicht alle Produktfunktionen realisiert werden. **Einflussfaktoren** *O1.1 : Time-To-Market* *O1.2 : Auslieferung von Produktfunktionen* *O2.1 : Anzahl Entwickler*
Lösung **Strategie: *Schrittweiser Ausbau*** System wird schrittweise ausgebaut. Anforderungen werden in der Reihenfolge ihrer Priorität realisiert. **Strategie: *Einkaufen statt Selbstentwickeln*** Einbindung externer COTS-Komponenten. **Strategie: *Wiederverwendung*** Einbindung interner wiederverwendbarer Komponenten.

Gebäudearchitektur





In der Gebäudearchitektur ist es üblich, von einem zu bauenden Gebäude verschiedene Modelle zu erstellen, bevor das Gebäude tatsächlich gebaut wird. Teilweise werden die Modelle als Prototypen benutzt, um dem zukünftigen Hausbesitzer einen Eindruck von der Ästhetik oder Funktion zu vermitteln. Teilweise sind die Modelle Pläne, wie es nachher tatsächlich werden soll. Es wird dabei stets eine Vielzahl unterschiedlicher Modelle erstellt, die jeweils bestimmte Aspekte betonen und von anderen abstrahieren. Der Grund liegt darin, dass unterschiedliche Personen, die am Gebäudebau beteiligt sind, das Gebäude aus unterschiedlichen Perspektiven betrachten wollen. Der spätere Besitzer interessiert sich für das Gesamterscheinungsbild, der Maurer für den Gebäuderahmen und der Elektriker für Kabelschächte, Steckdosen, Schalter etc. Das Prinzip, Unterschiedliches auseinander zu halten, ist in der Softwaretechnik als *Separation of Concerns* bekannt: die Trennung unterschiedlicher Belange. Dieses Prinzip wird auch beim Softwarearchitekturentwurf angewandt. Man nennt diese unterschiedlichen Baupläne oft *Sichten der Architektur*.

Während sich in der Gebäudearchitektur über viele Jahre feste Normen zu Bauzeichnungen, das heißt, Architektursichten, etabliert haben, stehen wir in der Softwaretechnik noch ganz am Anfang einer Entwicklung hin zu normierten Architektursichten. Ein erster Meilenstand auf diesem Weg ist der IEEE-Standard IEEE P1471 (2002) *Recommended Practice for Architectural Description of Software-intensive Systems*. Er stellt Richtlinien auf, wie man Sichten der Softwarearchitektur beschreiben soll. Da aber noch Uneinigkeit herrscht, welche Sichten der Softwarearchitektur überhaupt relevant sind, legt er uns nicht auf eine feste Menge solcher Sichten fest. Er geht viel mehr einen indirekten Weg. Er überlässt es uns, zu entscheiden, welche Sichten wir beschreiben wollen, fordert aber, dass für alle unsere Sichten definiert ist, was sie bedeuten.

Architektursichten und -blickwinkel

Definition

Architektursicht (View): Repräsentation eines ganzen Systems aus der Perspektive einer kohärenten Menge von Anliegen (IEEE P1471 2002).

Definition

Architekturblickwinkel (Viewpoint): Spezifikation der Regeln und Konventionen, um eine Architektursicht zu konstruieren und zu benutzen (IEEE P1471 2002).
Ein Blickwinkel ist ein Muster oder eine Vorlage, von der aus individuelle Sichten entwickelt werden können, durch Festlegung von

- Zweck,
- adressierte Betrachter
- und Techniken für Erstellung, Gebrauch und Analyse.

Unterschiedliche Sichten helfen der Strukturierung: Separation of Concerns.

Dazu führt der Standard eine Trennung von konkretem Inhalt und Bedeutung einer Sicht ein. Eine **Architektursicht (im Englischen: View)** ist die Repräsentation eines ganzen Systems aus der Perspektive einer kohärenten Menge von Anliegen (IEEE P1471 2002). Eine Architektursicht beschreibt also immer ein bestimmtes System, im Sinne der Gebäudearchitektur ist das also zum Beispiel die Fassadenansicht des neuen Bürogebäudes in der Universitätsallee 15. In der Software ist die Beschreibung der Klassen und ihrer Abhängigkeiten der Banksoftware *PayMe* der Version 1.3 in Form eines Klassendiagramms eine Architektursicht. Damit man ein Klassendiagramm jedoch verstehen kann, muss beschrieben sein, welche Konstrukte in solch einem Diagramm auftauchen dürfen, was sie bedeuten und wie sie kombiniert werden können. Während das Klassendiagramm für *PayMe* spezifisch und damit nur für *PayMe* relevant ist, ist die Beschreibung, wie Klassendiagrammme an sich aussehen können und was sie bedeuten, übergreifend gültig für alle denkbaren Klassendiagramme unabhängig von den Systemen, die sie beschreiben.

Für die Beschreibung aller möglichen Klassendiagramme führt der IEEE-Standard den Begriff Architekturblickwinkel (im Englischen: Viewpoint) ein. Ein **Architekturblickwinkel (Viewpoint)** ist die Spezifikation der Regeln und Konventionen, um eine Architektursicht zu konstruieren und zu benutzen (IEEE P1471 2002). Ein Blickwinkel ist ein Muster oder eine Vorlage, von der aus individuelle Sichten entwickelt werden können, durch Festlegung des Zwecks, der adressierten Betrachter und der Techniken für die Erstellung, den Gebrauch und die Analyse aller Sichten, die durch den Blickwinkel spezifiziert werden.

Diese Trennung zwischen Sicht und Blickwinkel ist der eigentlich bedeutende Beitrag des genannten Standards. Das heißt für uns, wenn wir eine Architektur durch eine Sicht beschreiben wollen, müssen wir zunächst in Form des Blickwinkels die Bedeutung der Sicht spezifizieren.

Siemens-Blickwinkel (Hofmeister u. a. 2000)

- **Konzeptioneller Blickwinkel**: beschreibt logische Struktur des Systems; abstrahiert weitgehend von technologischen Details
- **Modulblickwinkel**: beschreibt die statische logische Struktur des Systems
- **Ausführungsblickwinkel**: beschreibt die dynamische logische Struktur des Systems
- **Code-Blickwinkel**: beschreibt die „anfassbaren“ Elemente des Systems (Quelldateien, Bibliotheken, ausführbare Dateien etc.)

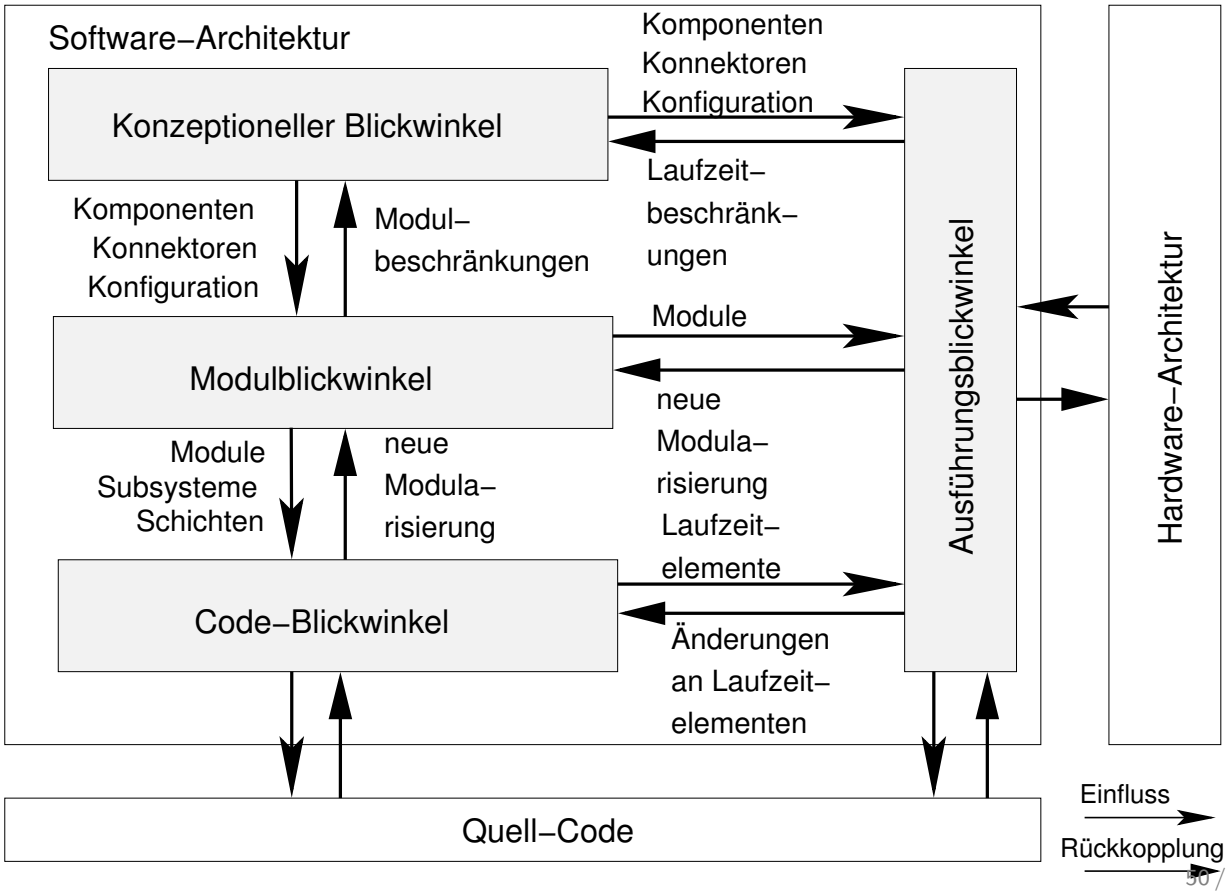
In der Literatur wurden sehr viele unterschiedliche Blickwinkel zur Beschreibung von Softwarearchitektur vorgeschlagen. Zachman (1987); Sowa und Zachman (1992); Zachman (1999) war einer der ersten Autoren, der Blickwinkel vorgeschlagen hat. Er schlug vor, Informationssysteme durch 6×6 verschiedene Blickwinkel zu beschreiben. Perry und Wolf (1992) haben mehrere Blickwinkel von Zachman zusammengefasst und die Anzahl der Blickwinkel damit auf drei reduziert, die letzten Endes aber äquivalent sind und lediglich unterschiedliche Aspekte hervorheben. Von Kruchten (1995) stammen die 4+1-Blickwinkel. Dazu gehören der logische Blickwinkel, der das System abstrakt und anwendungsnah beschreibt, der Prozessblickwinkel, der die dynamische Ausführung beschreibt sowie der statische Entwicklungsblickwinkel und der physikalische Blickwinkel, der das System in die Hardware einbettet. Diese vier Blickwinkel werden zusammengehalten durch einen redundanten Blickwinkel, der mittels Anwendungsszenarien beschreibt, wie die vier anderen Blickwinkel zusammenspielen.

Die unterschiedlichen Blickwinkel in der Literatur sind verwirrend, zumal sehr viele Blickwinkel unterschiedlicher Autoren sehr ähnlich sind. Etwas Ordnung hat hier das Buch von (Clements u. a. 2002) geschaffen, in dem Kategorien von Blickwinkeln beschrieben sind. Dieses Buch ist auch zu empfehlen für die Frage, wie man Softwarearchitektur dokumentiert.

Immerhin ist festzuhalten, dass viele der Blickwinkel sich auch auf die vier Siemens-Blickwinkel abbilden lassen (Hofmeister u. a. 2000). Es scheint, als ob mindestens die folgenden Aspekte zur Beschreibung eines Systems notwendig sind, die durch die vier Siemens-Blickwinkel abgedeckt werden:

- **Konzeptioneller Blickwinkel**: beschreibt die logische Struktur des Systems. Er abstrahiert weitgehend von technologischen Details wie z.B. konkrete Technologien, mit denen das System implementiert wird. Dieser Blickwinkel ist der Anwendungsdomäne am nächsten.
- **Modulblickwinkel**: beschreibt die statische logische Struktur des Systems in Form von Modulen und ihren Abhängigkeiten.
- **Ausführungsblickwinkel**: beschreibt die dynamische logische Struktur des Systems, das heißt, die Ausführung und Einbettung in die ausführende Hardware.
- **Code-Blickwinkel**: beschreibt die „anfassbaren“ Elemente des Systems (Quelldateien, Bibliotheken, ausführbare Dateien etc.).

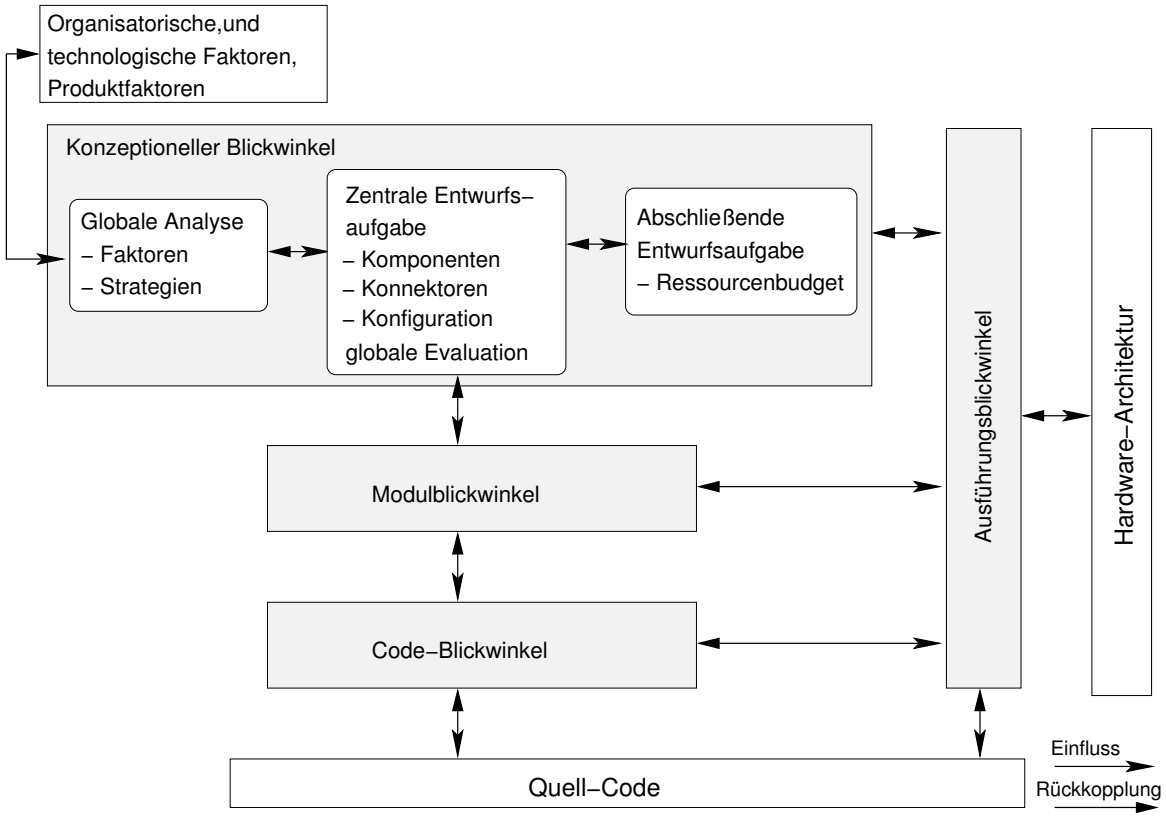
Siemens-Blickwinkel (Hofmeister u. a. 2000)



Die vier Siemens-Blickwinkel wurden durch Befragung praktizierender Softwarearchitekten bei Siemens ermittelt, haben also offenbar praktische Relevanz.

In der Methode von Hofmeister u. a. (2000) sind diese vier Blickwinkel der Gegenstand des Entwurfes. Die Softwarearchitektur wird entworfen, indem Sichten zu diesen vier Blickwinkeln erstellt werden.

Wir werden im Folgenden näher auf diese Blickwinkel eingehen, indem wir sie im Kontext der Hofmeister-Methode beschreiben.



Der Ausgangspunkt des Architekturentwurfes ist der konzeptionelle Blickwinkel. Mit Hilfe dieses Blickwinkels wird die logische Struktur der angestrebten Architektur beschrieben. Ähnlich wie beim Datenbankentwurf, wo wir zunächst von einem konzeptionelle Schema ausgehen, das die Daten beschreibt so wie sie in der Anwendungsdomäne sind. Es handelt sich noch nicht um den Entwurf einer konkreten Datenbank für das Anwendungsproblem, der ja letztlich stark von der verwendeten Technologie abhängt. In der Gebäudearchitektur gleicht der konzeptionelle Blickwinkel den ersten Skizzen, die der Architekt von dem späteren Haus macht. Dabei will er den Kopf noch frei haben von Detailentscheidungen, welche Heizung eingebaut wird etc.

Ausgehend von der globalen Analyse der Einflussfaktoren, entwirft der Softwarearchitekt die Hauptkomponenten und -konnektoren des Systems. Eine **Komponente** ist eine Berechnungseinheit oder eine Einheit, die Daten speichert. Dabei geht es nicht um einzelne Klassen oder gar spezifische Algorithmen und Datenstrukturen, sondern um die grobe Verteilung der Zuständigkeiten in einem System auf größere Bausteine. Wie umfangreich so ein Baustein ausfällt, hängt von der Größe des Systems ab. Je größer das System, desto größer werden die Komponenten initial ausfallen. Große Komponenten können dann später verfeinert werden. Ein Beispiel für eine Komponente in einem größeren System ist etwa eine Datenbank, in einer Client-Server-Architektur wären sowohl Client als auch Server erste Komponenten.

Komponenten interagieren miteinander, um eine gemeinsame Aufgabe zu erfüllen. Hierzu sind sie über so genannte **Konnektoren** verbunden. Ein Konnektor stellt den Kontroll- und Datenfluss zwischen Komponenten dar. Primitive Konnektoren sind Methodenaufrufe, Parameterübergabe und Zugriffe auf Attribute. Komplexere Konnektoren sind beispielsweise Mechanismen zur Interprozesskommunikation.

Die Verbindung der Komponenten durch Konnektoren ergibt die *Konfiguration*. Ist eine Konfiguration entstanden, kann sie durch ein Review begutachtet werden. Ein Evaluationskriterium ist zum Beispiel, ob die Komponenten und Konnektoren alle geforderten Anwendungsfälle auch tatsächlich unterstützen. Hierzu kann man auf dem Papier überlegen, was die Komponenten und Konnektoren alles machen würden für jeden einzelnen Anwendungsfall. Abschließend werden noch die Ressourcen für Komponenten und Konnektoren eingeplant, also wie viel Speicher und Rechenzeit Komponenten und Konnektoren zuerkannt bekommen.

Konzeptioneller Blickwinkel (Hofmeister u. a. 2000)

- ist der Anwendungsdomäne am nächsten
- Systemfunktionalität wird abgebildet auf
  - **Konzeptionelle Komponenten**: rechenbetonte Elemente oder Datenhaltung
  - **Konnektoren**: Kontroll- und Datenfluss zwischen konzeptionellen Komponenten

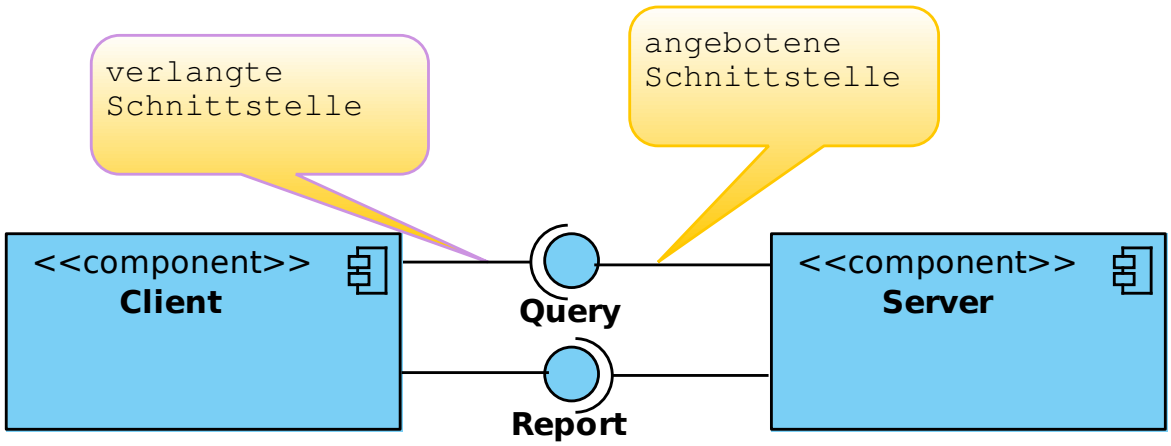
Engineering-Belange:

- Wie erfüllt das System seine Anforderungen?
- Wie werden Commercial-off-the-Shelf-Components (COTS) in das System integriert?
- Wie wird domänenspezifische Soft- und Hardware einbezogen?
- Wie kann die Auswirkung von Anforderungen oder der Anwendungsdomäne minimiert werden?

Wie gesagt, steckt hinter der Aufteilung in Blickwinkel das Prinzip der Trennung unterschiedlicher Belange. Welche Belange adressiert der konzeptionelle Blickwinkel also? Es geht dabei um die folgenden Fragen:

- Wie erfüllt das System seine Anforderungen?
- Wie werden Commercial-off-the-Shelf-Components (COTS) in das System integriert, falls dies geplant ist?
- Wie wird domänenspezifische Soft- und Hardware einbezogen?
- Wie kann die Auswirkung von Anforderungen oder der Anwendungsdomäne minimiert werden? Idealerweise sollte man nur eine Komponente anfassen müssen, wenn sich eine neue Anforderung aus der Anwendungsdomäne ergibt. Dazu muss man jedoch die möglichen Änderungen vorwegsehen und die Architektur entsprechend so entwerfen, dass es leicht fallen wird, die zukünftige Anforderung zu realisieren.

Konzeptionelle Sicht mit UML-Komponentendiagrammen



UML-Komponentendiagramme

Definition

Komponente: (engl. Component) verkapselte, eigenständige, vollständige und somit austauschbare Einheit.[1]

[1] Hier benutzt als logische Entwufskomponenten.

UML-Komponentendiagramme

Definition

Schnittstellen (engl. Interface): Menge von Operationen mit zugehörigem Protokoll:

- **angebotene Schnittstelle** (engl. provided Interface) einer Komponente K: Schnittstelle, die für andere bereit gestellt wird und von K implementiert wird.
- **verlangte Schnittstelle** (engl. required Interface) einer Komponente K: Schnittstelle, die K für das eigene Funktionieren voraussetzt.

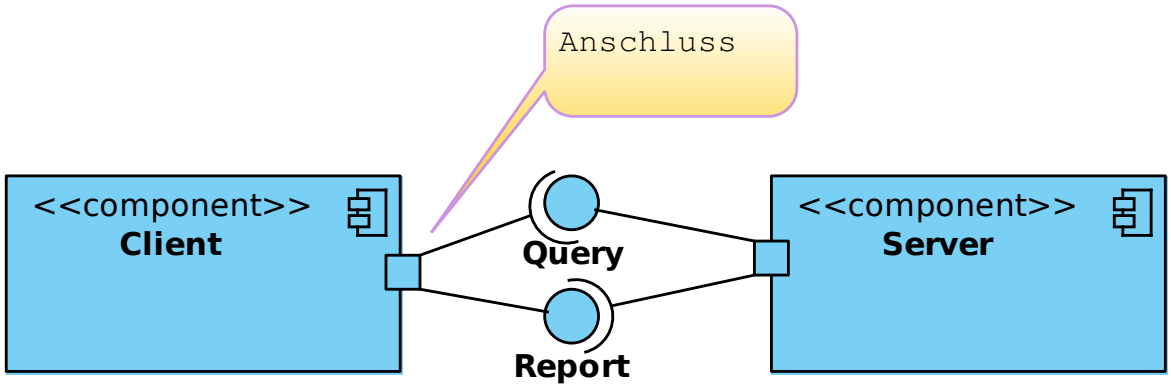
Definition

Protokoll: Spezifikation zulässiger Aufrufe mit Vor- und Nachbedingungen jedes Aufrufs.

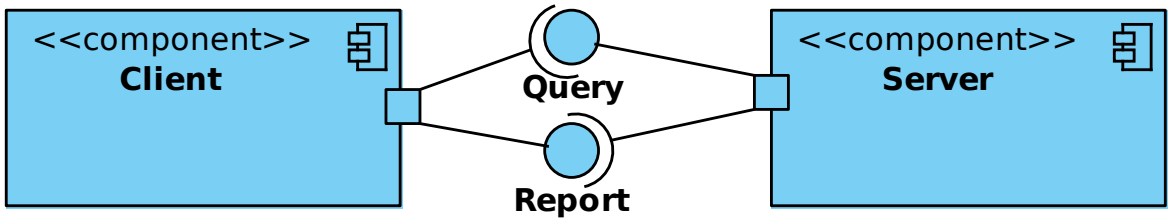
UML-Komponentendiagramme

Definition

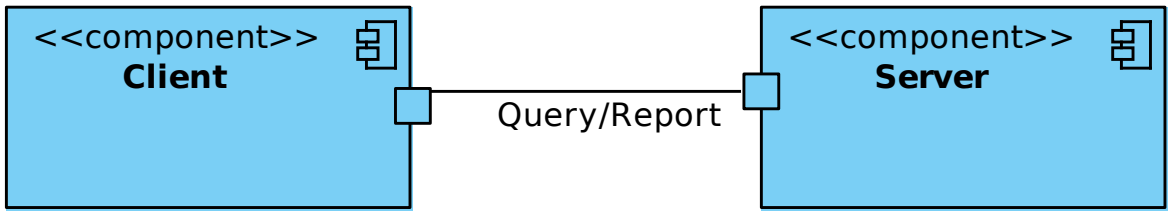
Anschluss stellt Möglichkeit des bidirektionalen Nachrichtenaustausches zwischen zwei zueinander passenden Rollen dar; umfasst angebotene (provided) und verlangte (required) Schnittstellen.



UML-Komponentendiagramme



. . . kann vereinfacht werden zu:



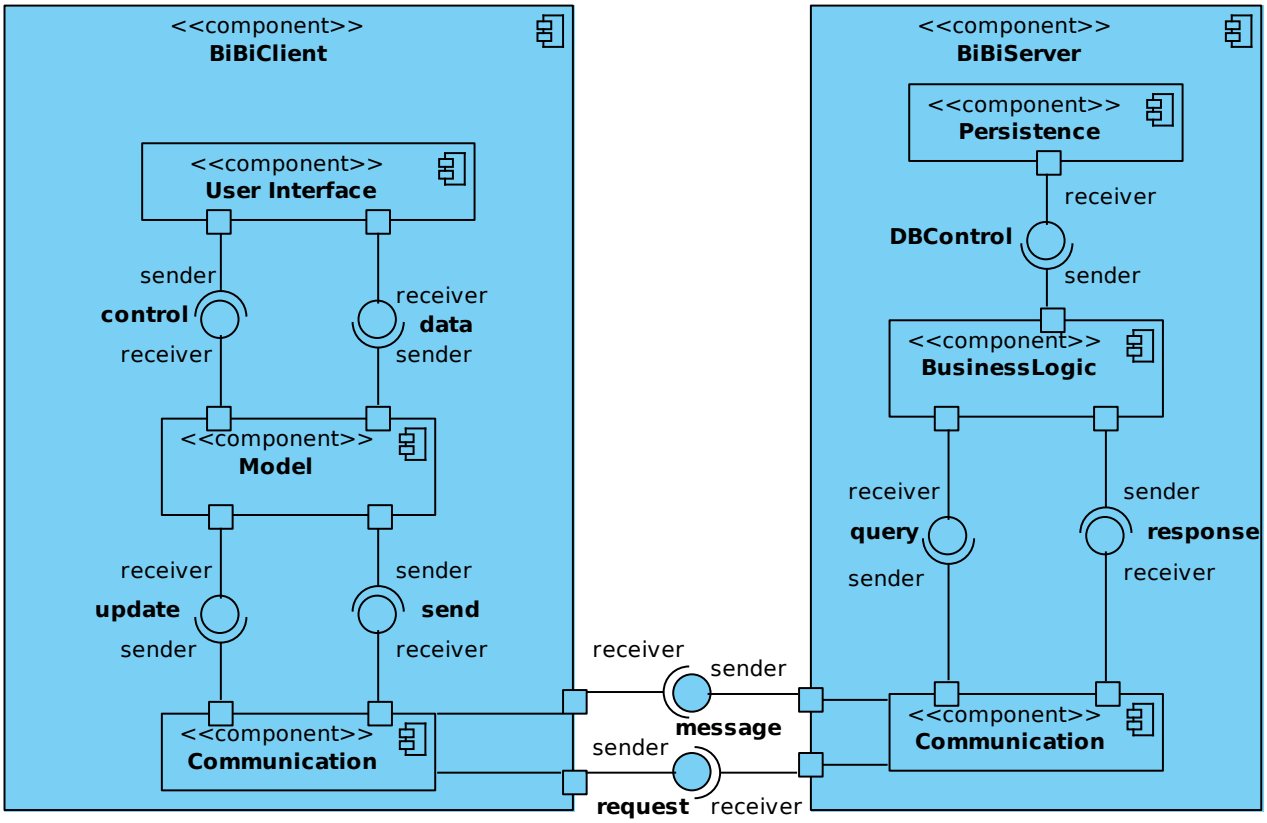
Spezialfall Relais: delegiert lediglich ein- und ausgehende Signale und ist in der UML eine (unstrukturierte) Klasse.

Varianten (Störrle 2005):

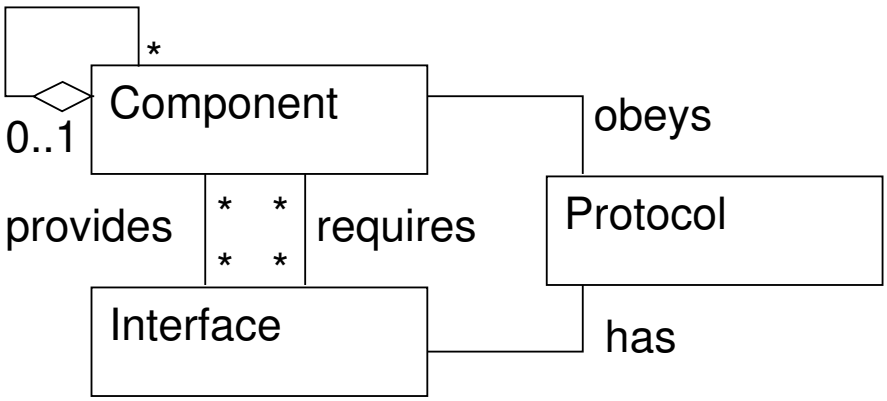
- innerer Anschluss: verläuft von Teil zu umschließender Komponente
  Konsequenz: Teil muss nach außen sichtbar sein
- Relais: steht zwischen Kontext und Anschlüssen von Teilen; delegiert Signale weiter, Teile bleiben verborgen; wird zusammen mit Delegierungsverbindern benutzt
- Transponderanschluss: steht zwischen Kontext und Anschlüssen mehrerer Teile; führt eigene Funktion aus (z.B. Umkodierung, Umleitung, Pufferung von Signalen etc.)

Wenn zwei Komponenten über Schnittstellen (provide/require) an der Erbringung einer Funktionalität zusammen wirken, können sie zu einem Anschluss (Port) zusammengefasst werden (nach Störrle (2005), Seite 101, Abb. 6.14).

Konzeptionelle Sicht von Bibi



Konzeptioneller Blickwinkel

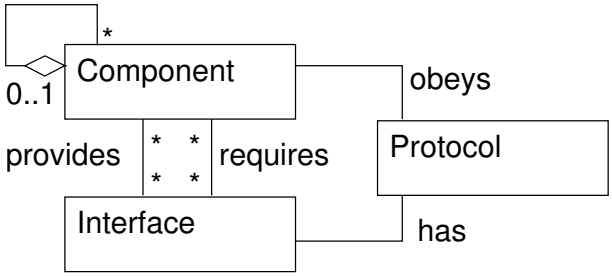


Nachdem wir nun die Rolle und den Zweck des konzeptionellen Blickwinkels kennen, betrachten wir näher, wie eine konzeptionelle Sicht aufgebaut sein kann. Wir wenden uns also der Frage zu, wie der konzeptionelle Blickwinkel eigentlich aussieht.

Wir haben bereits über die prinzipiellen Modellierungselemente gesprochen: Komponenten, Konnektoren/Schnittstellen und die Art und Weise, wie sie in einer Konfiguration angeordnet werden.

An dieser Stelle weichen wir hier etwas vom originalen konzeptionellen Blickwinkel von Hofmeister ab (Hofmeister u. a. 2000). Um entsprechende Sichten mit existierenden UML-Werkzeugen erstellen zu können, verwenden wir anstelle des Vorschlags von Hofmeister Komponentendiagramme der UML.

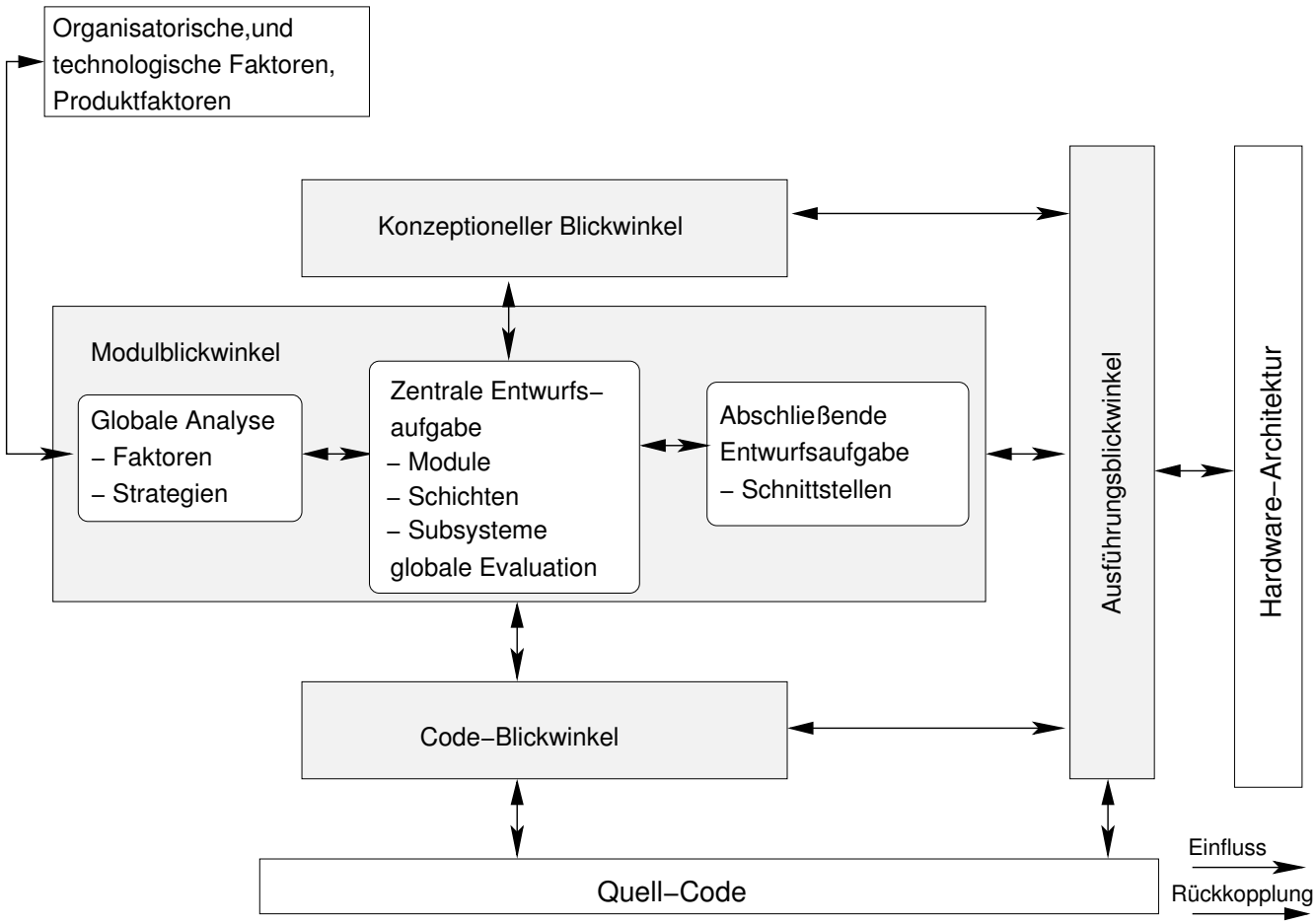
Das folgende Diagramm in UML beschreibt den konzeptionellen Blickwinkel genauer:



Wie wir sehen, besteht eine Konfiguration aus einer Menge von Komponenten und ihren Konnektoren/Schnittstellen. Komponenten können verfeinert werden, was die Komposition andeutet. Wenn eine Komponente verfeinert wird in Teilkomponenten, müssen diese selbstverständlich auch über Schnittstellen wieder verbunden werden.

Fragen

Wie können statische Aspekte (Struktur, Aufbau) einer konzeptionellen Sicht verfeinert werden?



Im nächsten Schritt überführen wir die konzeptionelle Sicht in die Modulsicht, die den statischen Aufbau des Systems beschreibt in Form von Modulen, Schnittstellen, Subsystemen und Schichten. Es ist gut möglich, dass sich beim Entwurf der konzeptionellen Sicht neue Aspekte ergeben haben, so dass wir die globale Analyse gegebenfalls wiederholen, bevor wir uns an die zentrale Entwurfsaufgabe machen, die Module zu entwerfen. Nach Entwurf der Module und ihre Anordnung in Schichten und Subsystemen, entwerfen wir abschließend deren Schnittstellen.

In diesem Abschnitt vertiefen wir den Modulsichtentwurf. Es sei vorausgeschickt, dass wir nicht streng sequentiell Modulsicht, dann Ausführungs- und dann Code-Sicht entwerfen. Vielmehr ist der Entwurfsprozess meist iterativ.

Modulblickwinkel

- bildet Komponenten und Konnektoren auf Module, Schichten und Subsysteme ab
- setzt konzeptionelle Lösung mit aktuellen Softwareplattformen (Programmiersprachen, Bibliotheken) und Technologien um
- beschreibt statische Aspekte

Engineering-Belange:

- Wie wird das Produkt auf die Software-Plattform abgebildet?
- Welche Softwaredienste werden benutzt und von wem?
- Wie wird das Testen unterstützt?
- Wie können Abhängigkeiten zwischen Modulen minimiert werden?
- Wie kann die Wiederverwendung von Modulen maximiert werden?
- Welche Techniken können eingesetzt werden, um Auswirkungen von Änderungen zu minimieren?

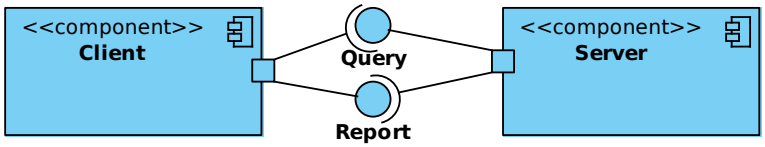
Beim Entwurf der Modulsicht bilden wir die Komponenten und Konnektoren der konzeptionellen Sicht auf Module, Schichten und Subsysteme ab. Hierzu setzen wir die konzeptionelle Lösung mit der aktuellen Softwareplattform (Programmiersprachen, Bibliotheken, Werkzeuge) und Technologien um. Dabei werden rein statische Aspekte beschrieben.

Die Belange, die durch die statische Modulsicht adressiert werden sind die folgenden:

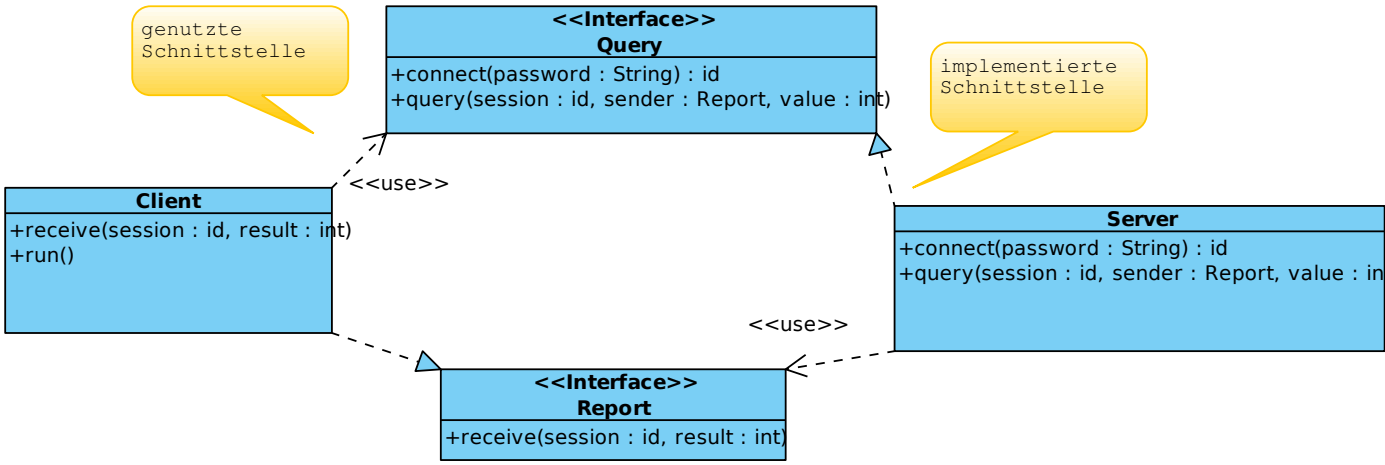
- Wie wird das Produkt auf die Software-Plattform abgebildet?
- Welche Softwaredienste werden benutzt und von wem?
- Wie wird das Testen unterstützt?
- Wie können Abhängigkeiten zwischen Modulen minimiert werden?
- Wie kann die Wiederverwendung von Modulen maximiert werden?
- Welche Techniken können eingesetzt werden, um Auswirkungen von Änderungen zu minimieren?

Für die Modulsichten interessieren sich Programmierer im Hinblick auf die Programmierung. Tester interessieren sich für die Modulsicht im Hinblick auf den Blackbox-Komponententest sowie den Integrationstest. Ein Projektleiter kann von der Modulsicht Gebrauch machen, um festzulegen, welcher Entwickler welches Modul implementieren soll.

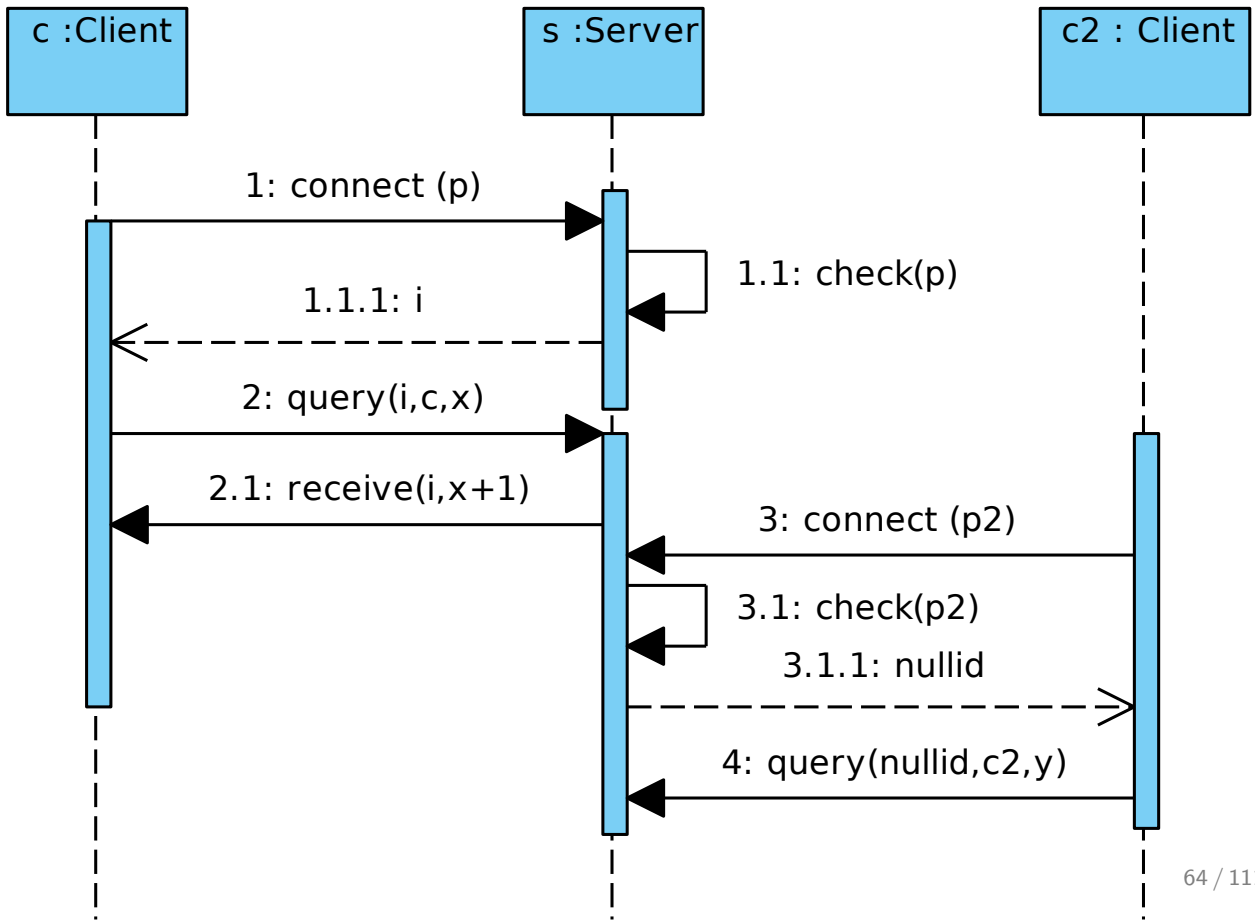
Vom Abstrakten zum Konkreten



. . . atomare Komponenten können mit Klassendiagramm konkretisiert werden:

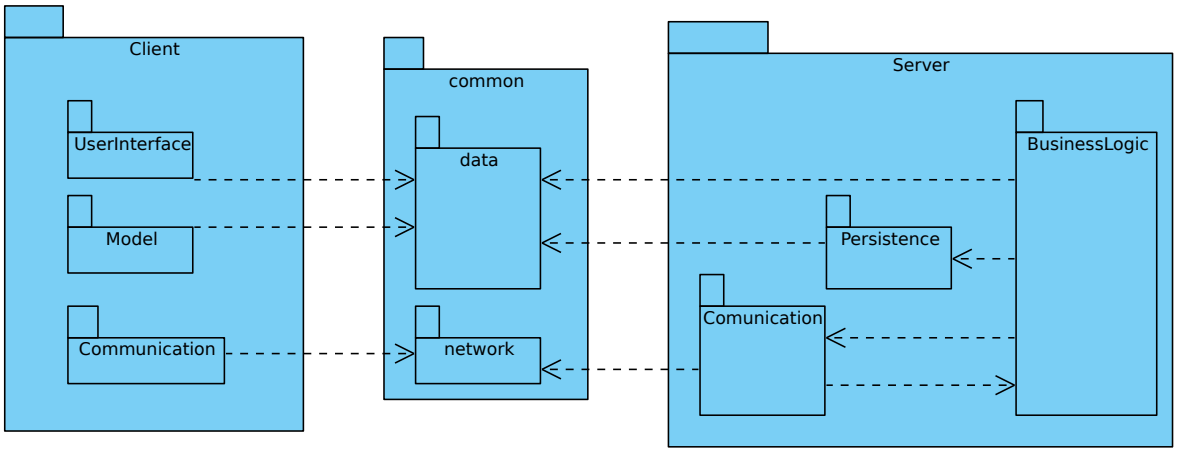


Sequenzdiagramm für das Protokoll



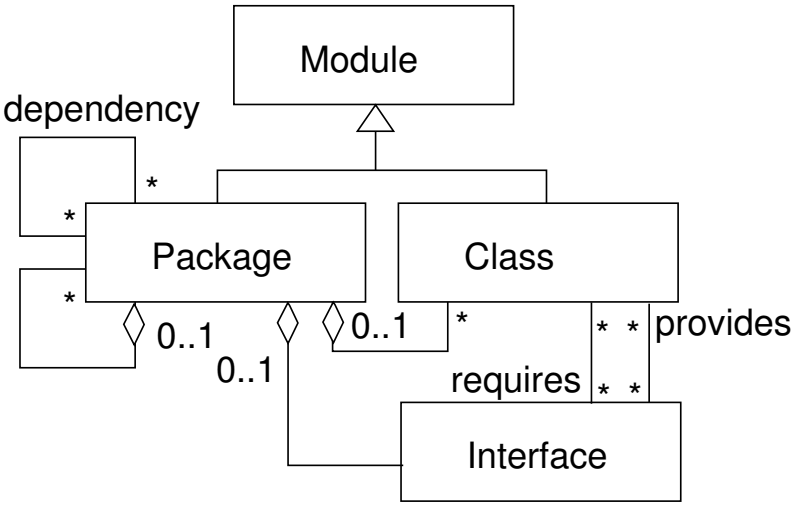
Grobe Modulsicht von Bibi

. . . zusammengesetzte Komponenten können mit Paketdiagrammen konkretisiert werden (die ihrerseits wieder durch Paket- und Klassendiagramme verfeinert werden):

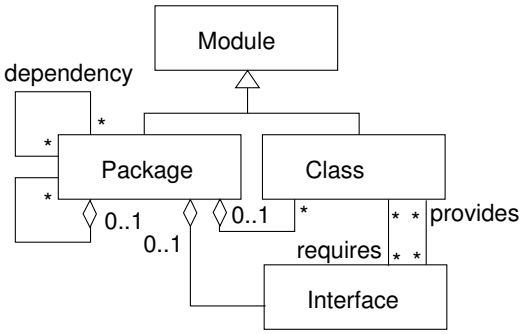


Modulblickwinkel (Hofmeister u. a. 2000)

. . . mit Hilfe von Paket- und Klassendiagrammen



Der Modulblickwinkel wird durch das folgende UML-Diagramm beschrieben (auch hier weichen wir leicht von der Definition von Hofmeister u. a. (2000) ab):



Ein Modul ist eine statische, austauschbare Gliederungseinheit mit einem bestimmten Zweck. Es kann in der Beschreibung durch Teilmodule verfeinert werden. Ein Modul hat zwei Kategorien von Schnittstellen: die Schnittstellen, die es selbst anderen anbietet (*provides*) und jene, die es von anderen benötigt (*require*). In den meisten Programmiersprachen kann man nur die zur Verfügung gestellten Schnittstellen angeben, aber nicht jene, die das Modul voraussetzt. Wenn Module aber ausgetauscht werden und die Auswirkung von Änderungen analysiert werden sollen, dann benötigt man auch Wissen darüber, welche Schnittstelle ein Modul voraussetzt.

Wir unterscheiden in diesem Blickwinkel zwei unterschiedliche Arten von Modulen (auch hierin weichen wir von dem originalen Blickwinkel von Hofmeister ab). Sie unterscheiden sich durch ihre Granularität. Die feinste Gliederung ist die Klasse, die man im Architekturentwurf nicht weiter verfeinern würde (erst im Feinentwurf). Klassen können zu Paketen zusammen gefasst werden. Hier folgen wir also dem Gliederungskonzept von Java. Auch Pakete können zu größeren Paketen zusammengefasst werden. Zur Modellierung der statischen Modulsicht verwenden wir Klassen- und Paketdiagramme der UML. Darin können wir zwar Klassen Schnittstellen zuordnen, aber nicht Paketen. Die Schnittstellen von Paketen ergeben sich implizit durch die darin enthaltenen Klassen. Enthält ein Paket A eine Klasse X, die eine Klasse Y benutzt, die in einem Paket B enthalten ist, so existiert dann und nur dann eine *Dependency* zwischen A und B.

Module sind ein etwas allgemeineres Konzept als Klassen. Klassen sind ein Konzept einer Programmiersprache und des objektorientierten Paradigmas. Module können aber in C auch durch Dateien implementiert werden. Oft werden sie durch mehrere Dateien oder Klassen implementiert. Insofern hat das Konzept *Modul* seine Berechtigung.

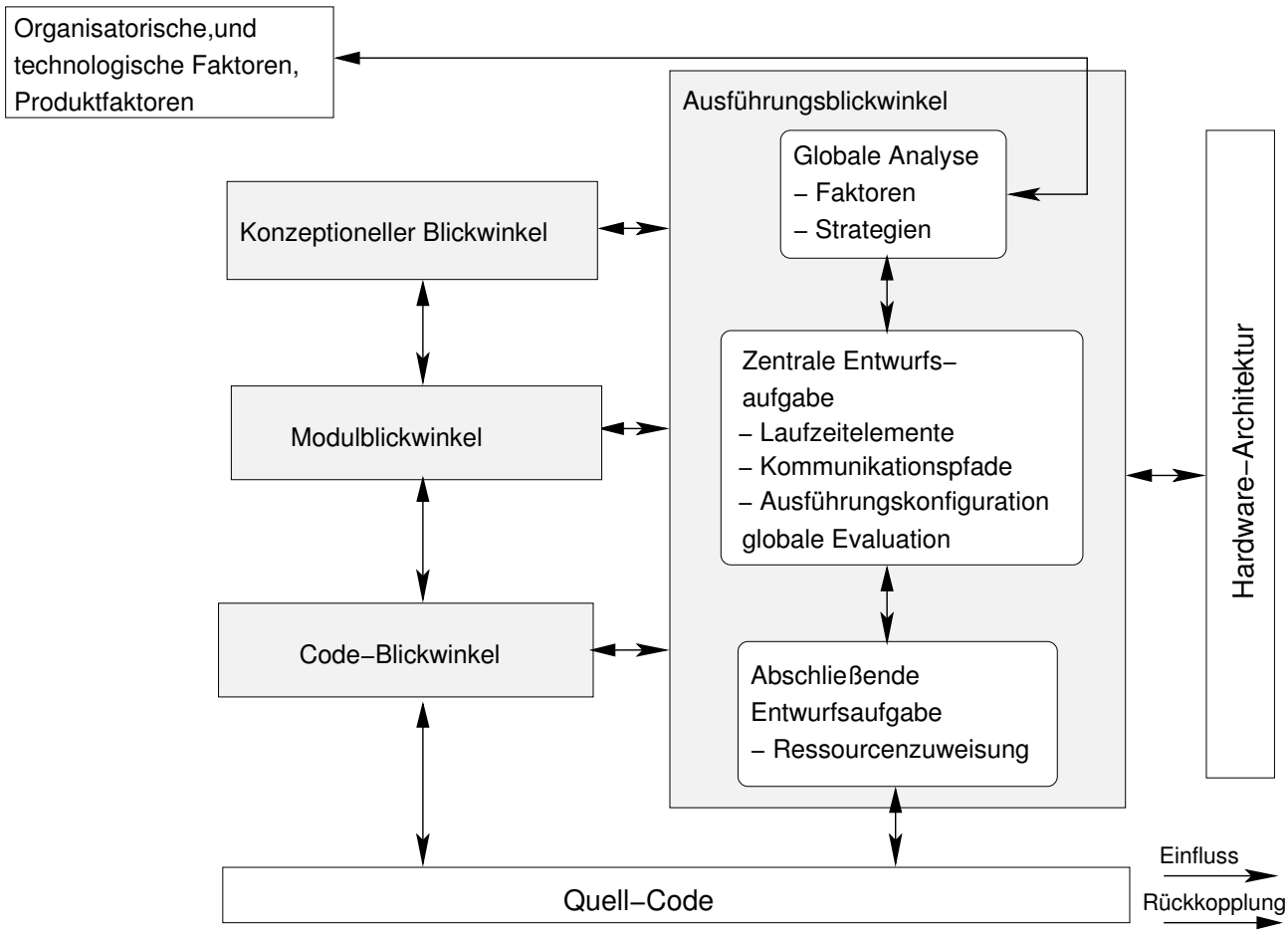
Weil Module aber Klassen ähneln, kann man für sie auch eine ähnliche Notation verwenden. Die UML-Notation für sichtbare und außerhalb der Vererbungshierarchie unsichtbare (*protected*) Elemente können zur Spezifikation der *provides*-Schnittstelle verwendet werden. Alternativ kann man auch auf die UML-Darstellungen für Schnittstellen (*Lolipop*) zurück greifen.

Man kann weitere Konzepte von Modulen ableiten und so den Modulblickwinkel erweitern. Man könnte zum Beispiel ein Subsystemkonzept oder Schichten einführen. In diesem Falle würde man von *Package* ableiten. Die erweiterte Semantik des neuen Konzepts würde man in den Diagrammen durch Stereotypen deutlich machen.

Bei einem strikten Schichtenmodell dürfen Module einer Schicht nur auf Module der nächst tieferen Schicht zugreifen. Dies erhöht die Austauschbarkeit. Ein Schicht implementiert eine Art virtuelle Maschine. Auch Schichten haben beide Arten von Schnittstellen und können wieder weiter in Teilschichten verfeinert werden. Zu Schichten folgen später im Kontext von Architekturstilen noch mehr Details.

Fragen

Wie kann man die Struktur des Systems zur Laufzeit beschreiben?



Software ist primär Verhalten. Durch die Modulsicht wird aber nur der innere strukturelle Aufbau beschrieben. Im nächsten Schritt kümmern wir uns um dynamische Aspekte, also Dinge, die zur Laufzeit relevant sind. Dazu bilden wir die Module auf Elemente der Ausführungsplattform (sowohl Hardware als auch Software) ab. Dazu dient der Ausführungsblickwinkel.

Ausführungsblickwinkel

- beschreibt dynamische Aspekte
- beschreibt, wie Module auf Elemente der Ausführungs- und Hardwareplattform abgebildet werden
- definiert Laufzeitelemente und deren Attribute (Speicher- und Zeitbedarf, Allokation auf Hardware)

Engineering-Belange:

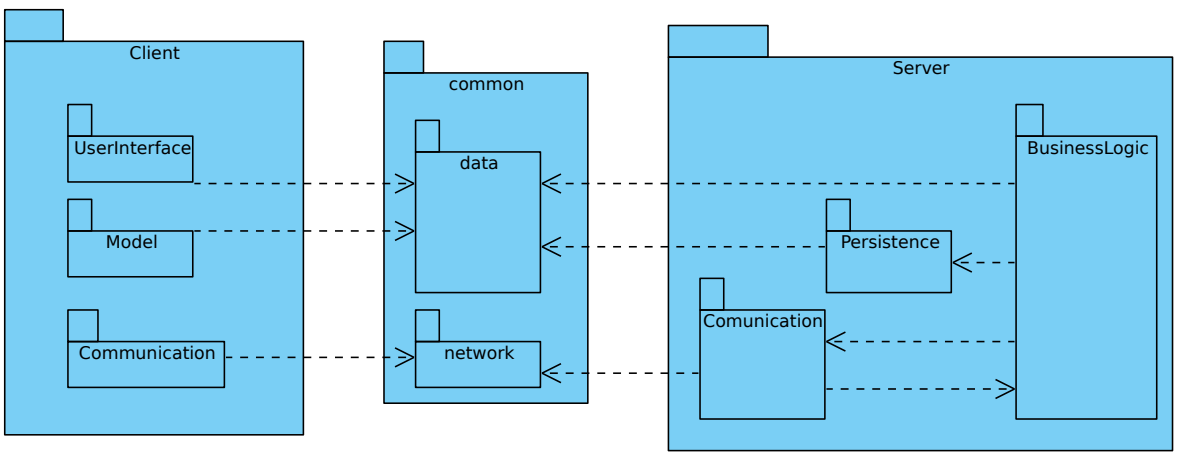
- Wie verläuft Kontroll- und Datenfluss zwischen Laufzeitkomponenten?
- Wie kann Ressourcenverbrauch ausgewogen werden?
- Wie können Nebenläufigkeit, Replikation und Verteilung erreicht werden, ohne die Algorithmen unnötig zu verkomplizieren?
- Wie können Auswirkungen von Änderungen in der Ausführungsplattform minimiert werden?

Der Ausführungsblickwinkel definiert die Laufzeitelemente und deren Attribute wie Speicher- und Zeitbedarf und Allokation auf Hardware. Wir beschreiben hierbei, welche Prozesse und Objekte existieren, wie sie miteinander zur Ausführungszeit kommunizieren und wie sie auf die ausführende Hardware abgebildet werden.

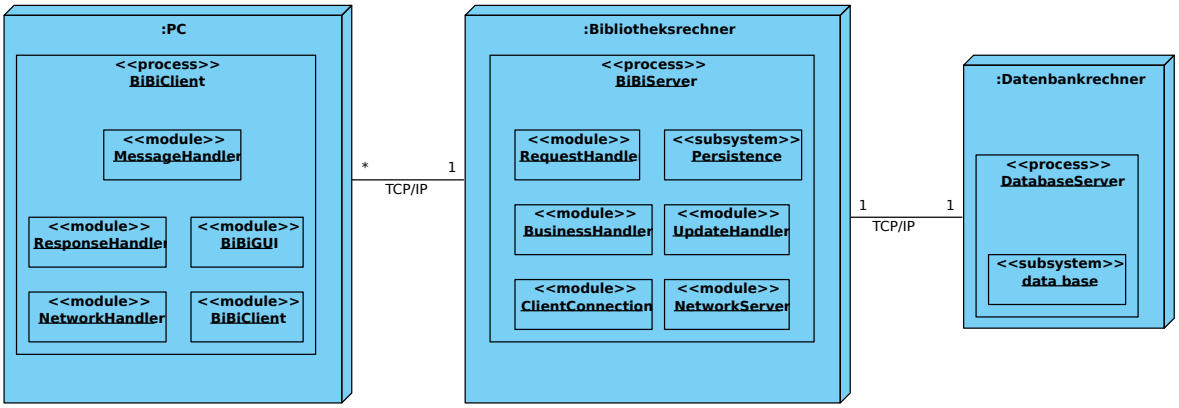
Die adressierten Belange des Ausführungsblickwinkels sind damit:

- Wie verläuft der Kontroll- und Datenfluss zwischen den Laufzeitkomponenten (Prozesse, Objekte etc.)?
- Wie kann der Ressourcenverbrauch (Speicher, Rechenzeit etc.) ausgewogen werden?
- Wie können Nebenläufigkeit, Replikation und Verteilung erreicht werden, ohne die Algorithmen unnötig zu verkomplizieren?
- Wie können Auswirkungen von Änderungen in der Ausführungsplattform minimiert werden (also Änderungen der Hardware)?

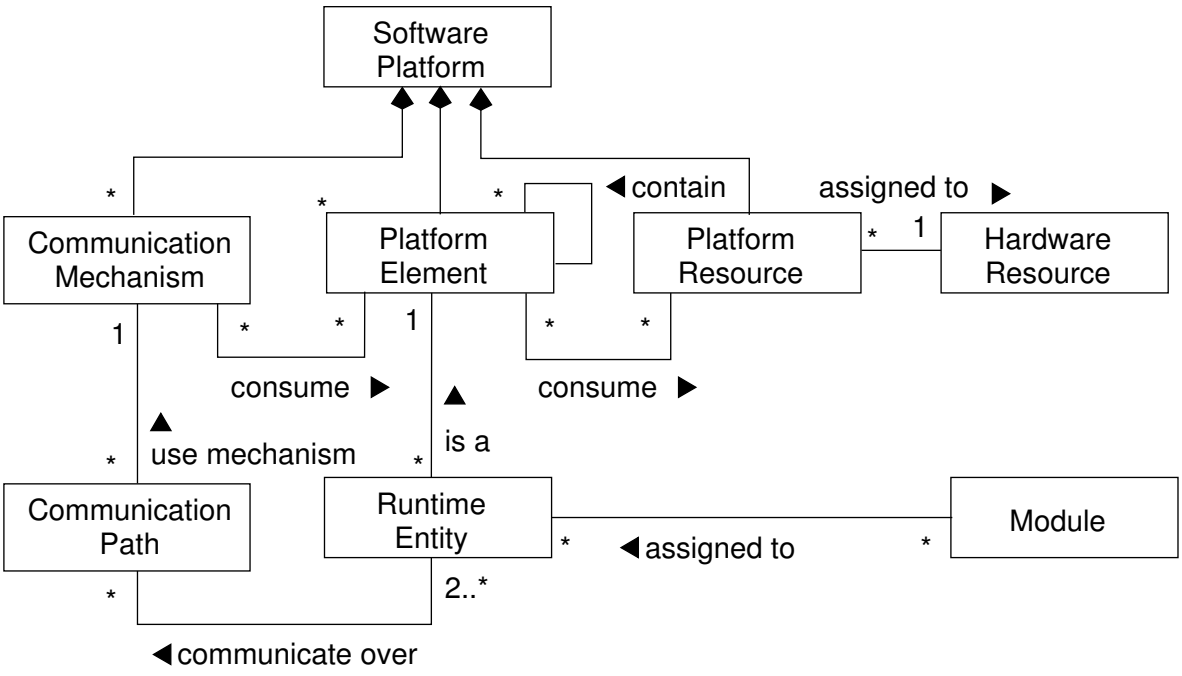
Grobe Modulsicht von Bibi



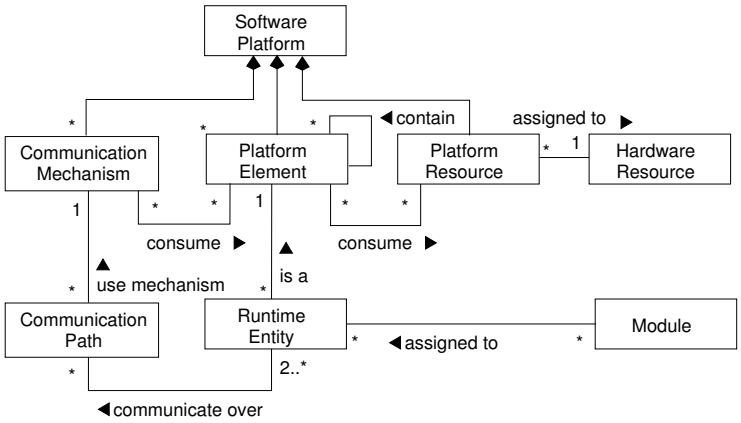
Ausführungssicht von Bibi



Ausführungsblickwinkel (Hofmeister u. a. 2000)



Das folgende UML-Diagramm beschreibt den Ausführungsblickwinkel:



Die Software-Plattform hält alle Aspekte der Ausführung zusammen. Sie besteht zunächst aus Plattformelementen, dies sind Dinge, die zur Laufzeit angelegt werden und ausgeführt werden können (z.B. Prozesse; wir lernen weitere Beispiele weiter unten kennen).

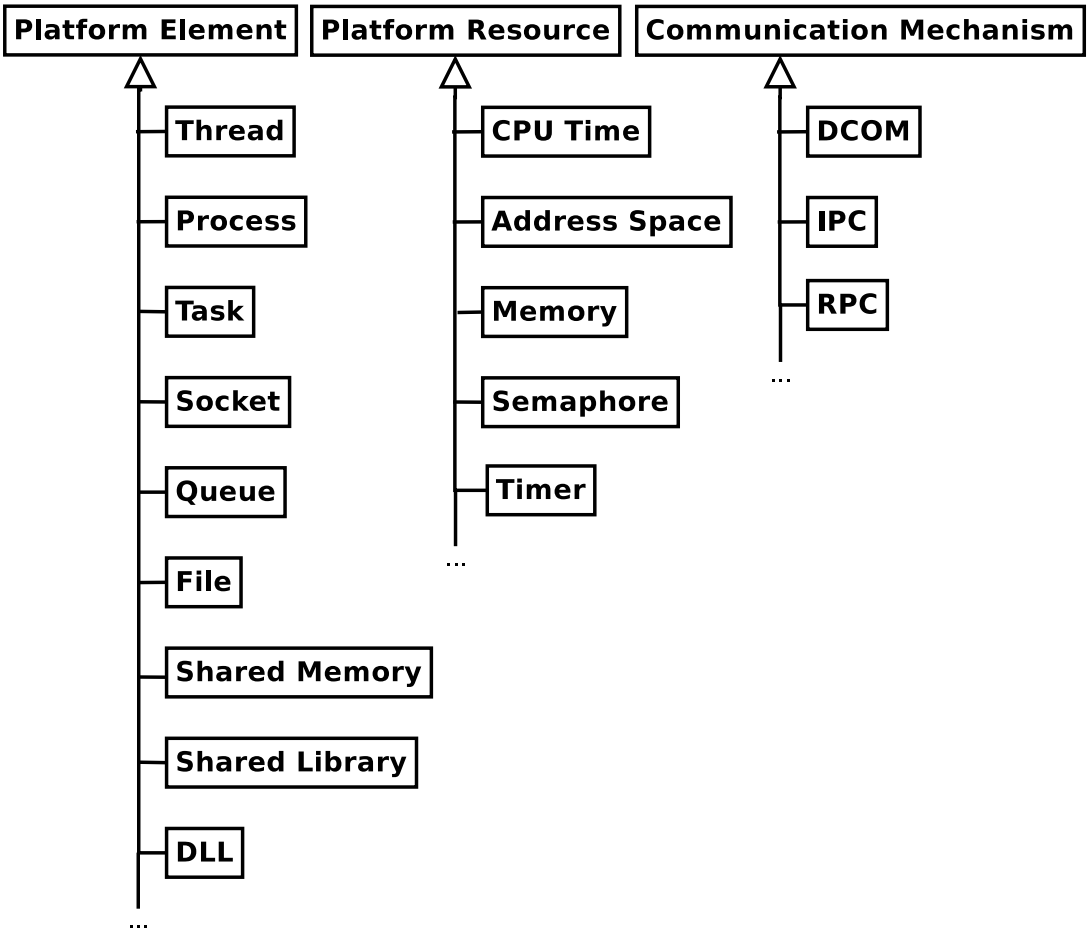
Die Plattformelemente haben Instanzen, die Laufzeiteinheiten (*Runtime Entity*). Die Laufzeiteinheiten kommunizieren über Kommunikationspfade (*Communication Path*) miteinander. Mindestens zwei konkrete Laufzeitinstanzen kommunizieren dabei. Die Kommunikationspfade verwenden hierzu einen Kommunikationsmechanismus, wie beispielsweise eine *Remote Method Invocation* in Java oder eine Corba-Interprozesskommunikation.

Bei der Ausführung konsumiert eine Laufzeiteinheit Laufzeitressourcen wie Speicher oder Rechenzeit.

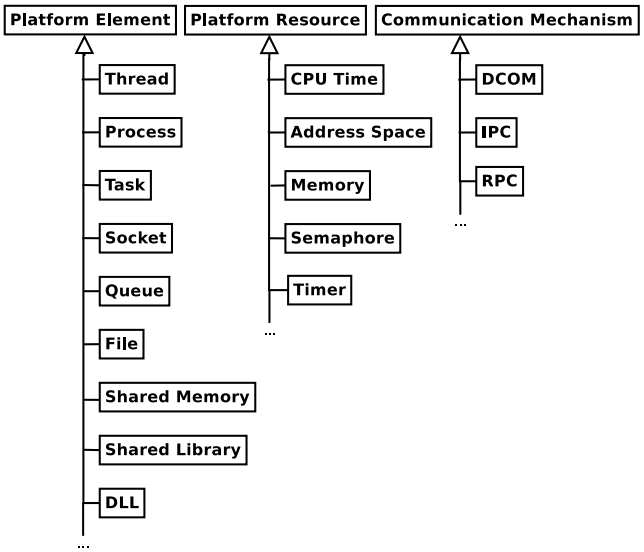
Auf der rechten Seite des Diagramms sehen wir die Einbettung in die Hardwareumgebung und in die statische Modulsicht. Module sind den Laufzeiteinheiten, die sie ausführen, zugeordnet. Dabei kann ein Modul natürlich von vielen Laufzeiteinheiten verwendet werden. Durch diese Bezüge zur Modulsicht werden die beiden Sichten miteinander verknüpft.

Die Plattformressourcen werden auf konkrete Hardware-Ressourcen abgebildet, die sie zur Verfügung stellen, wie beispielsweise Prozessoren oder gemeinsamen Speicher.

Ausführungsblickwinkel (Hofmeister u. a. 2000)



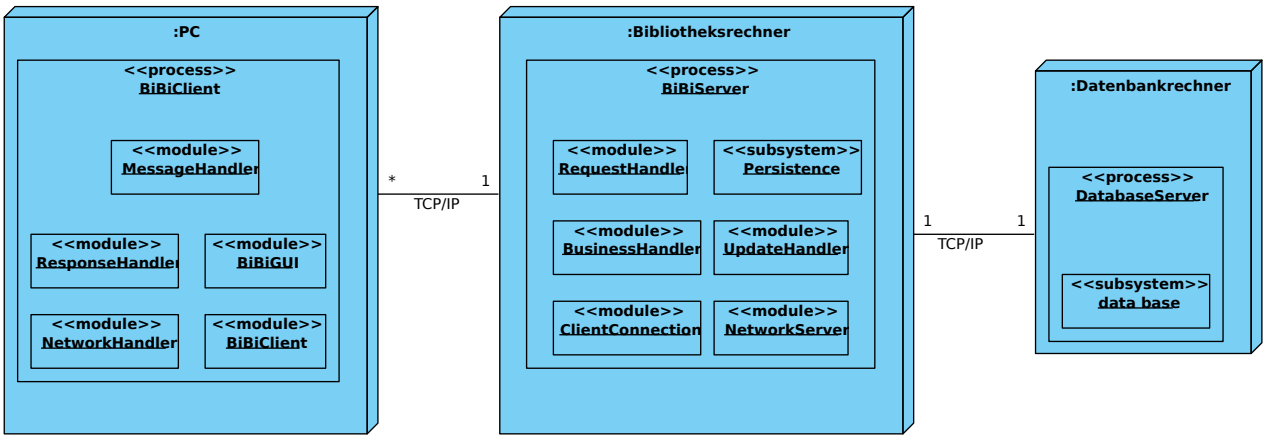
Die Konzepte des Ausführungsblickwinkels sind noch relativ abstrakt. Die folgende Vererbungshierarchie konkretisiert die Konzepte. Ein Plattformelement ist eine Softwareeinheit, die zur Laufzeit existiert. Dazu gehören zum Beispiel Objekte als Instanzen von Klassen. Für die Realisierung von paralleler Ausführung haben wir Threads (als leichtgewichtige Prozesse mit gemeinsamem Speicher), Prozesse mit getrenntem Speicher oder Tasks (als eigenes Sprachkonstrukt aus einer Programmiersprache). Diese Prozesse kommunizieren über weitere Plattformelemente wie Sockets, Warteschlangen, Dateien oder geteiltem Speicher. Außerdem kann man Module bündeln in dynamischen Bibliotheken und auf unterschiedlichen Rechnern installieren.



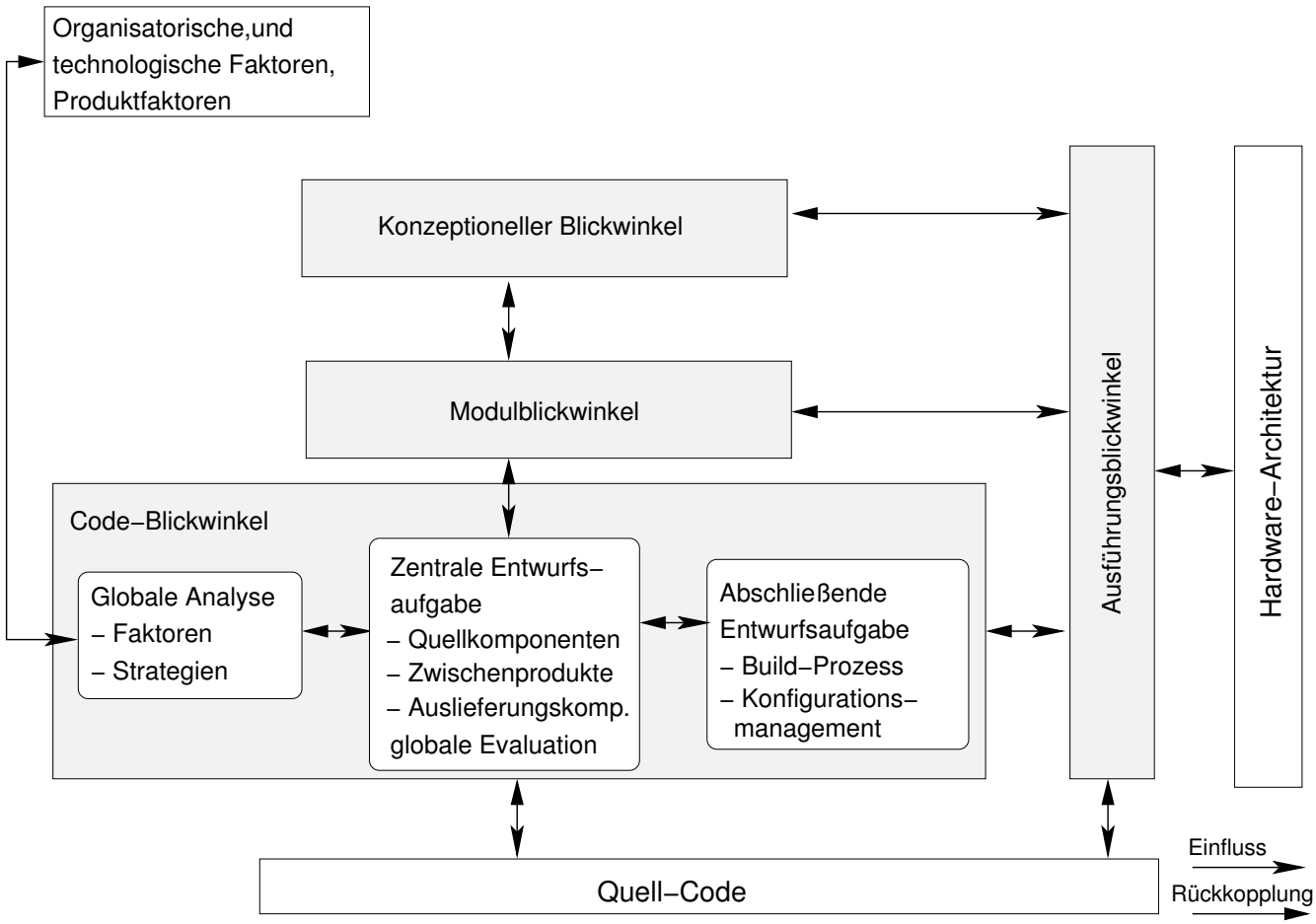
Plattformressourcen sind Ressourcen, die uns von der Software- und Hardwareplattform zur Verfügung gestellt werden. Sie werden von den Plattformelementen konsumiert, während diese ihre Arbeit erledigen. Dazu gehören neben Speicher und CPU-Zeit Betriebsmittel für die Synchronisation wie Semaphore oder Timer.

Kommunikationsmechanismen schließlich dienen den Plattformelementen zum Nachrichtenaustausch. Unter dem Betriebssystem Windows gibt es hierfür zum Beispiel DCOM. Plattformunabhängigere Mechanismen sind die Interprozesskommunikation (IPC) oder der Remote Procedure Call (RPC), bei Java auch bekannt als Remote Method Invocation (RMI).

Für Diagramme, die die Ausführungssicht beschreiben, gibt es keine Standardnotation. Die Diagrammart in der UML, die der Ausführungssicht am nächsten kommt, ist das Deployment-Diagramm. Es zeigt aber nicht alle Aspekte, die im Ausführungsblickwinkel nach Hofmeister u. a. (2000) möglich sind. Hofmeister u. a. (2000) schlagen deshalb eine eigene Notation vor, die an das Deployment-Diagramm der UML angelehnt ist. Das folgende Diagramm ist ein Beispiel hierfür:



Wir erkennen darin drei Rechnertypen, die als Quader dargestellt sind. In ihnen geschachtelt tauchen drei Prozesstypen auf, die mit dem Stereotyp *process* gekennzeichnet sind. Die Schachtelung drückt aus, dass der geschachtelte Prozess auf dem jeweiligen Rechner ausgeführt wird. Die Module, die die Prozesse dabei ausführen, sind in den Prozessen geschachtelt. Die Kommunikationspfade werden als Assoziationen zwischen den Prozessen dargestellt. Die Beschriftung der Assoziation gibt den Kommunikationsmechanismus wieder. Die Multiplizitäten an den Kommunikationspfaden geben an, wie viele Prozesse jedes Typs an der Kommunikation beteiligt sind. In unserem Fall ist das je eine n:1-Beziehungen, d.h., mehrere Clients kommunizieren mit einem Server und genau ein Server kommuniziert mit genau einem Datenbank-Server.



Nachdem wir Modulsicht und Ausführungssicht entworfen haben, planen wir die konkrete Implementierung. Elemente aus der Modulsicht und Ausführungssicht sind abstrakte Konzepte, die dann in einer Programmiersprache realisiert werden müssen. Dies geschieht, indem Programmierer Dateien anlegen, in denen der Implementierungscode (textuell oder graphisch) aufgeführt ist. Wir nennen dies die **Quellkomponenten**. Aus den Quellkomponenten wird dann - möglicherweise über Zwischenprodukte - das erzeugt, was schließlich als Code an den Kunden ausgeliefert wird. Beispielsweise haben wir unser Programm in C geschrieben. Dann erzeugt der Compiler daraus als Zwischenprodukt Objektdateien und möglicherweise Bibliotheken. Daraus erzeugt der Linker schließlich das ausführbare Programm. Das ausführbare Programm und alle dynamischen Bibliotheken stellen die Ausführungskomponenten dar. Da Dateien meist die Einheit sind, in der Aufgaben an Programmierer verteilt werden, und die meisten Versionskontrollsysteme nur ganze Dateien verwalten, bestimmt die Aufteilung in Dateien die Aufgabenverteilung und das Konfigurationsmanagement. Außerdem kann der Build-Prozess unter Umständen bei großen Systemen, die nicht selten in vielen unterschiedlichen Programmiersprachen geschrieben werden, sehr komplex werden. Der Build-Prozess benötigt nicht selten viel Zeit — selbst auf mächtiger Hardware. Aus diesen Gründen sollte für mittlere und große Systeme die Abbildung der abstrakten Konzepte aus der Modulsicht und Ausführungssicht auf "anfassbare" Dateien und auch der Build-Prozess geplant und dokumentiert werden. Dies ist die Aufgabe der Code-Sicht.

Bevor wir die Code-Sicht entwerfen, ist es möglicherweise notwendig, die globale Analyse nochmals zu wiederholen, wenn sich durch den Entwurf der vorherigen Sichten Einflussfaktoren und Strategien geändert haben sollten.

Code-Blickwinkel

- bildet Laufzeiteinheiten auf installierbare Objekte (ausführbare Dateien, dynamische Link-Bibliotheken etc.) ab
- bildet Module auf Quellkomponenten (Dateien) ab
- zeigt, wie die installierbaren Dateien aus den Quellkomponenten generiert werden

Engineering-Belange:

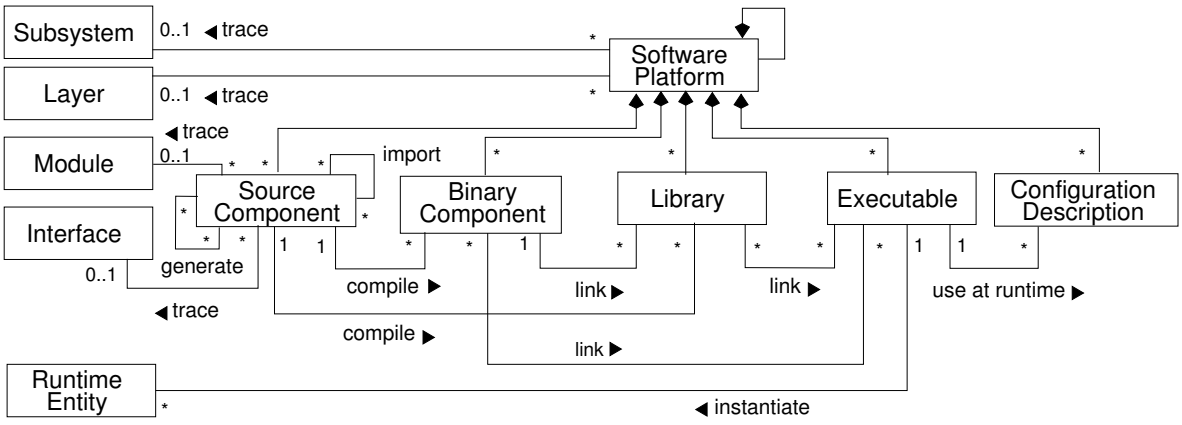
- Wie kann die Zeit und der Aufwand für Produkt-Upgrades verringert werden?
- Wie sollen Produktversionen verwaltet werden?
- Wie kann die Build-Zeit verringert werden?
- Welche Werkzeuge werden in der Entwicklungsumgebung benötigt?
- Wie wird Integration und Test unterstützt?

Die Code-Sicht bildet die Laufzeiteinheiten auf installierbare Objekte (ausführbare Dateien, dynamische Link-Bibliotheken etc.) ab. Außerdem bildet sie Module auf Quellkomponenten (Dateien) ab und zeigt, wie die installierbaren Dateien aus den Quellkomponenten generiert werden.

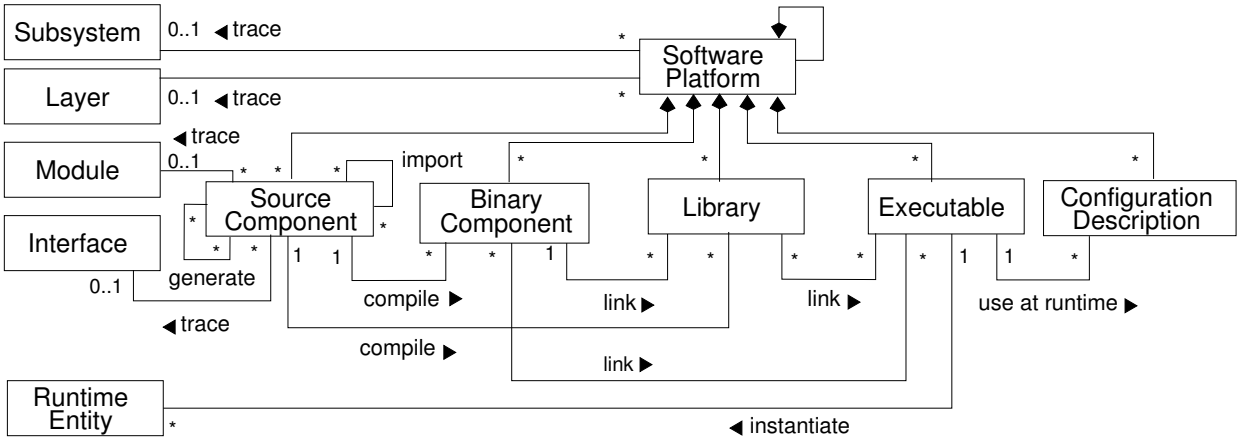
Die Belange, die mit der Code-Sicht adressiert werden, sind wie folgt:

- Wie können die Zeit und der Aufwand für Produkt-Upgrades verringert werden? Dazu müssen wir wissen, welche ausgelieferten Einheiten wir erneuern müssen. Im schlimmsten Fall muss der Kunde alles neu installieren. Dies wollen wir aber vermeiden.
- Wie sollen Produktversionen verwaltet werden? Unter Umständen müssen wir nachvollziehen können, welche einzelnen Dateien für die Installation bei einem bestimmten Kunden eingeflossen sind, um zum Beispiel Fehler zu diagnostizieren. Diese Information muss durch das Konfigurationsmanagement bereit gehalten werden.
- Wie kann die Build-Zeit verringert werden? Die Zeit für die vollständige Übersetzung kann erheblich sein und damit die Entwicklung stark hemmen. Bei großen Systemen kann es viele Stunden dauern, bis ein System vollständig übersetzt ist. Wir wollen deshalb die Abhängigkeiten zwischen den Dateien möglichst gering halten, um den Build-Prozess so kurz wie möglich zu halten.
- Welche Werkzeuge werden in der Entwicklungsumgebung benötigt? In manchen Fällen können zum Beispiel Teile des Code automatisch generiert werden.
- Wie wird Integration und Test unterstützt? Bei der inkrementellen Integration werden Systemteile sukzessiv hinzugenommen und in ihrer Interaktion getestet. Dazu muss man den inneren Aufbau und die Abhängigkeiten aus der Modulsicht kennen und wissen, welche Dateien dazu jeweils zu übersetzen und zu testen sind.

Code-Blickwinkel (Hofmeister u. a. 2000)



Code-Sichten werden durch den folgenden Code-Blickwinkel beschrieben:

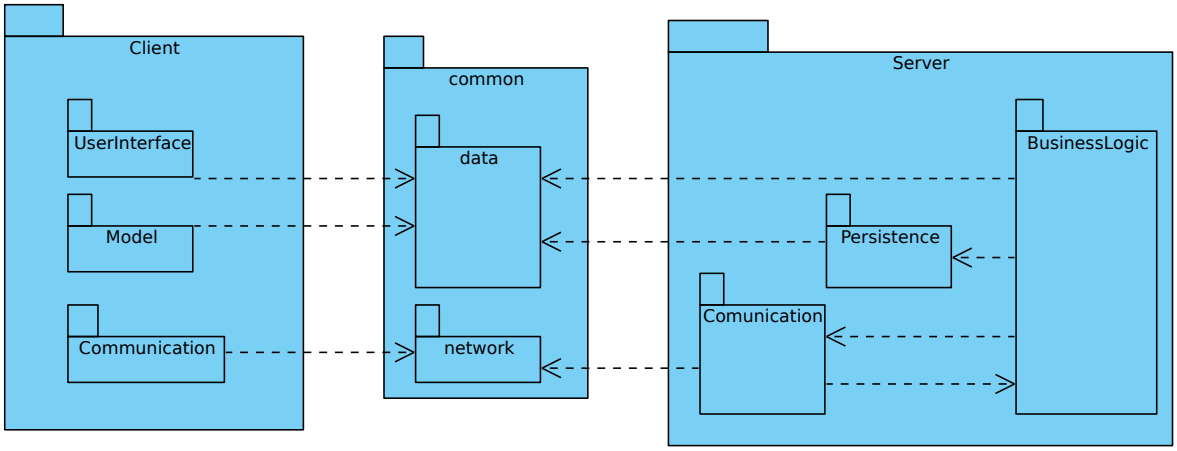


Alles, was für die Konstruktion des Systems notwendig ist, wird versammelt unter der Software-Plattform (Software Platform). Dazu gehören primär die Quell-Komponenten (Source Component), die die Module und Schnittstellen implementieren. Das sind in Java beispielsweise die Java-Quelldateien. Sie hängen zum Teil voneinander ab, indem sie sich importieren. In Java gibt es dafür die `import`-Anweisung. Außer reinen Code-Dateien betrachten wir auch zum Beispiel UML-Diagramme als Quell-Komponenten, wenn aus ihnen Quell-Code generiert wird. Die Quell-Komponenten werden bei kompilierten Sprachen von einem Compiler in eine Binärform übersetzt. In Java ist das der Java-Bytecode. Die Binärdateien werden in einem weiteren Schritt oft in statischen oder dynamischen Bibliotheken zusammengefasst. In Java sind das die JAR-Bibliotheken. Ein Linker linkt die Binärdateien und Bibliotheken schließlich zu einem ausführbaren Programm. Das Linken kann jedoch auch erst zum Zeitpunkt des Programms stattfinden. In diesem Fall spricht man von einem dynamischen Linker. Die ausführbaren Programme können häufig durch Konfigurationsdateien (Configuration Description) zur Start- oder Laufzeit konfiguriert werden. Beispielsweise kann die Sprache, in der mit dem Anwender kommuniziert wird, bei internationalisierten Programmen eingestellt werden.

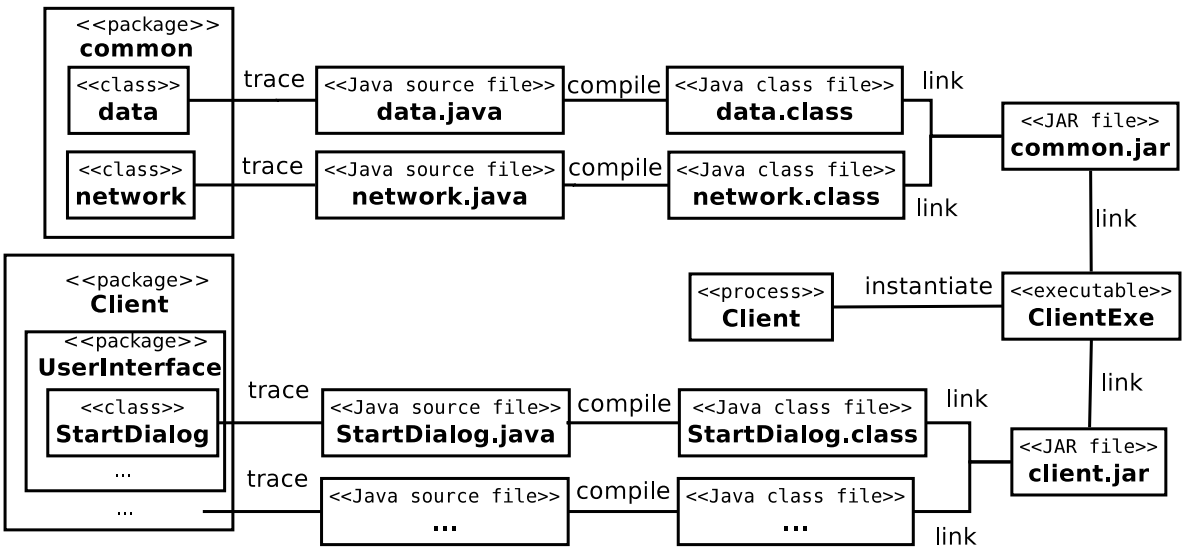
Auf der linken Seite des Diagramms zum Code-Blickwinkel finden wir Elemente des statischen Modulblickwinkels und des Ausführungsblickwinkels. Auf diese Weise werden die verschiedenen Sichten miteinander verbunden. Zwischen Quell-Komponenten und Modulen und Schnittstellen haben wir eine *trace*-Beziehung. Sie beschreibt, welche Module durch welche Dateien implementiert werden, und ermöglicht damit die Nachvollziehbarkeit (im Englischen: Traceability) zwischen unterschiedlichen Ebenen eines Systems. Schichten und Subsysteme sind meist keiner einzelnen Quell-Komponente zuzuordnen. Vielmehr bilden wir sie ab auf die Software-Plattform. Eine Schicht oder ein Subsystem implementiert damit eine Software-Plattform.

Die Programme erzeugen (*instantiate*) während ihrer Ausführung die Laufzeiteinheiten (*Runtime Entity*).

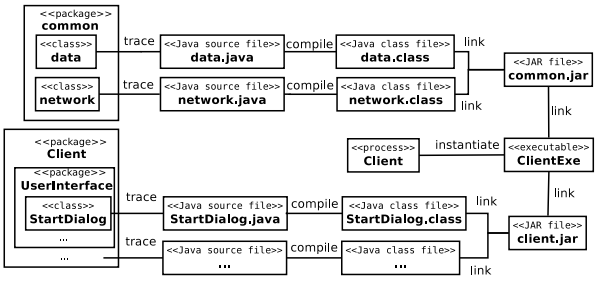
Grobe Modulsicht von Bibi



Code-Blickwinkel: Beispiel



Das folgende Diagramm ist ein Beispiel für eine Code-Sicht:



Die Teilmodule *data* und *network* des Pakets *common* werden durch die Dateien *data.java* und *network.java* umgesetzt. Das zusammengesetzte Modul *common* wird nicht unmittelbar durch eine Quell-Komponente implementiert. Vielmehr besteht es aus den oben genannten Teilmodulen, die bestimmten Quell-Komponenten zugeordnet sind. Hierarchische Module werden als reine Gliederungseinheiten verwendet. Die Moduldekomposition ergibt einen Baum. Nur die Blätter dieses Baumes stellen Module dar, für die auch Code geschrieben werden muss.

Die aus den Java-Dateien generierten Class-Dateien werden in einer JAR-Bibliothek für den Client abgelegt. Bei Ausführung der Bibliothek, indem die entsprechende *main*-Methode aktiviert wird, wird dann der Client-Prozess erzeugt. Dazu benötigt man die beiden JAR-Dateien *common.jar* und *client.jar*.

Der Architekt hat sich entschieden, die Netzwerkmodule in ein eigenes JAR-Archiv abzulegen, damit dasselbe Archiv sowohl für die Clients als auch für den Server verwendet werden kann.

Das Modell von Hofmeister ist sehr stark an herkömmlichen Sprachen wie C oder C++ orientiert. Im Falle von Java sind einige Spezialisierungen beziehungsweise Anpassungen sinnvoll.

Eine Erleichterung in Java ist der Umstand, dass eine Class-Datei mit genau einer Java-Quelldatei korrespondiert (und umgekehrt; außer für innere Klassen). Und die Package-Struktur korrespondiert mit der physischen Dateisystem-Struktur.

Die Bibliotheken in Java heißen JAR-Dateien (Java Archive; im Wesentlichen nichts anderes als ein Zip-Archiv mit Java-Class-Dateien). Die Class-Dateien werden nicht zu den Bibliotheken gelinkt, sondern lediglich hinzugefügt.

Es gibt in Java eigentlich kein *Exectuable* wie bei C oder C++. Jede Klasse, die eine statische Methode *main* hat, kann zum Hauptprogramm werden. Enthält eine JAR-Datei mindestens eine solche Klasse, kann sie als *Executable* betrachtet werden. Eine JAR-Datei (oder auch eine Class-Datei) *instantiates* eine *Runtime Entity*, wenn ihr *main* aufgerufen wird, oder wenn ein Thread aktiviert wird.

Code-Blickwinkel: Beispiel

Tabellendarstellung:

Module	Source File	JAR
data	data.java	common.jar
network	network.java	common.jar
StartDialog	StartDialog.java	client.jar
...	...	...

JAR	instantiates
client.jar	Client
...	...

In der Praxis wird man kaum für die in der Code-Sicht auftretenden Relationen ein UML-Diagramm zeichnen, da es viel zu viele sind. Einfacher beschreibt man sie durch Tabellen, wie zum Beispiel:

Module	Source File	JAR
data	data.java	common.jar
network	network.java	common.jar
StartDialog	StartDialog.java	client.jar
...	...	...

JAR	instantiates
client.jar	Client
...	...

Wenn Namenskonventionen existieren (im Falle von Java heißt die Class-Datei, die zu einer Java-Datei `myfile.java` gehört, schlicht `myfile.class`, wenn es sich nicht um eine innere Klasse handelt) müssen nicht alle Relationen durch Tabellen beschrieben werden.

Solche Zusammenhänge werden meist in Build-Werkzeugen wie *make*, *Ant* oder *Maven* festgehalten.

Was ist Modularisierung?

Definition

Modularisierung: Dekomposition eines Systems in Module.
Modul: austauschbares Programmteil, das eine geschlossene Funktionseinheit bildet.
Arbeitspaket: von einer Person oder einer kleinen Gruppe von Entwicklern entwerfbar, implementierbar, testbar, verstehbar, änderbar, ...

Conways Gesetz:

Conways Gesetz:

Die Struktur der Software spiegelt die Struktur der Organisation wider.

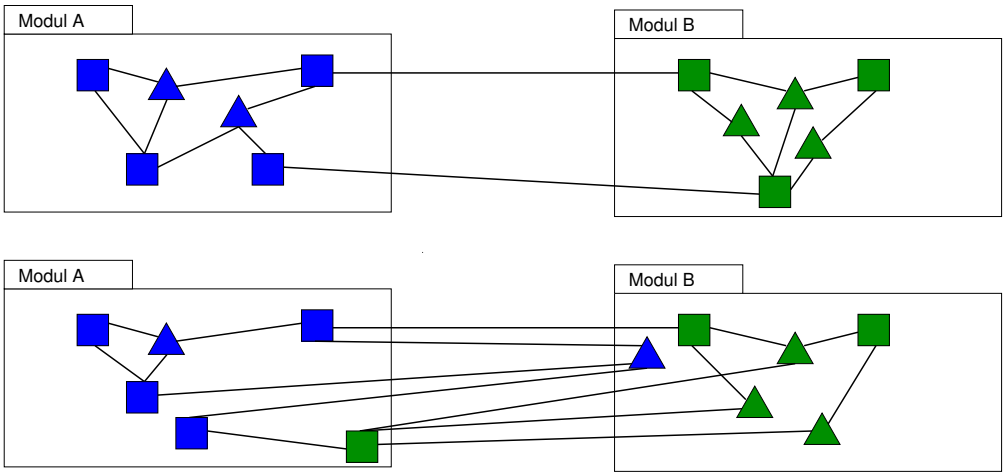
Kopplung und Zusammenhalt

Definition

Kopplung: Grad der Abhängigkeit zwischen Modulen.

Definition

Zusammenhalt (Kohärenz): Verwandtschaft der Teile eines Moduls.



Der Architekt eines Konzertsaals bemüht sich, den Saal so zu bauen, dass die akustische Störung von außen extrem gering ist, die Hörbarkeit im Saal dagegen extrem hoch. Dem entspricht bei der Arbeit des Software-Architekten die Gliederung in Module so, dass

- die Kopplung (d.h. die Breite und Komplexität der Schnittstellen) zwischen den Modulen möglichst gering,
- der Zusammenhalt (d.h. die Verwandtschaft zwischen den Teilen eines Moduls) möglichst hoch wird.

Das Konzept von Kopplung und Zusammenhalt basiert ursprünglich auf den FORTRAN- und COBOL-Programmen der frühen 70'er Jahre. Heute haben wir Sprachen, die uns mehrere Abstraktionsebenen bieten. Dabei streben wir auf der Modulebene vor allem niedrige Kopplung (bei mäßigem Zusammenhalt), auf der Prozedurebene vor allem hohen Zusammenhalt (bei erheblicher Kopplung) an.

Stufen des Zusammenhalts (von schlecht nach gut)

Stufe	**innerhalb einer Funktion/Moduls**	**betrifft**
kein Zusammenhalt	rein zufällige Zusammenstellung	Module
Ähnlichkeit	z.B. ähnlicher Zweck, also etwa alleMatrixoperationen; auch Fehlerbehandlung	Module
zeitliche Nähe	Verwendung zum selben Zeitpunkt, z.B. Initialisierung, Abschluss	Module, Funktionen
gemeinsame Daten	Zugriff auf bestimmte Daten, exklusiv, z.B. Kalenderpaket (Systemuhr)	Funktionen, Module
Hersteller / Verbraucher	ein Teil erzeugt, was der andere verwendet	Module, Funktionen
einziges Datum	Kapselung einer Datenstruktur, Zusammenfassung der Operationen	Module, Funktionen
einzige Operation	Operation, die nicht mehr zerlegbar ist	Funktionen

Stufen der Kopplung (von schlecht nach gut)

Stufe	**zwischen Funktionen/Modulen**	**betrifft**
Einbruch	Veränderung des Codes	Funktionen
volle Öffnung	Zugriff auf alle Daten, z.B. auf alle Attribute einer Klasse	(Module) Funktionen
selektive Öffnung	bestimmte Attribute sind zugänglich oderglobal durch expliziten Export/Import	Module, Funktionen
Funktions-kopplung	nur Funktionsaufruf und Parameter	Module (Funktion)
keine Kopplung	Es besteht keine logische Beziehung (Zugriff ist syntaktisch unmöglich)	Module

Kriterien für einen guten Software-Entwurf

Jeder Entwurf ist ein Kompromiss.

- Geringe Kopplung zwischen allen Modulen
- Hoher Zusammenhalt in allen Funktionen und Modulen
- Kriterium der naiven Suche: Jede Einheit sollte einen leicht erkennbaren Sinn haben.
- Abstraktion: Implementierungsdetails verbergen
- Kapselung von Dingen, die sich ändern können (Geheimnisprinzip; Information Hiding)
- Etwa gleich große Einheiten (Module, Funktionen). Abweichungen sollten plausibel sein; generell dürfen einfache Objekte größer sein als komplizierte.

Kriterien für einen guten Software-Entwurf (Forts.)

- Uniforme Entwurfsstrategie; wenn z.B. eine Schichtenstruktur gewählt wurde, sollte sie nicht an irgendeiner Stelle korrumpiert sein.
- Uniforme Gestaltung der Schnittstellen.
- Uniforme Benennung.
- Prinzip der Isomorphie zur Realität (Michael Jackson[2]): Die Software sollte der Realität strukturell gleichen, damit die Änderungen der Realität in der Software leicht nachvollzogen werden können.

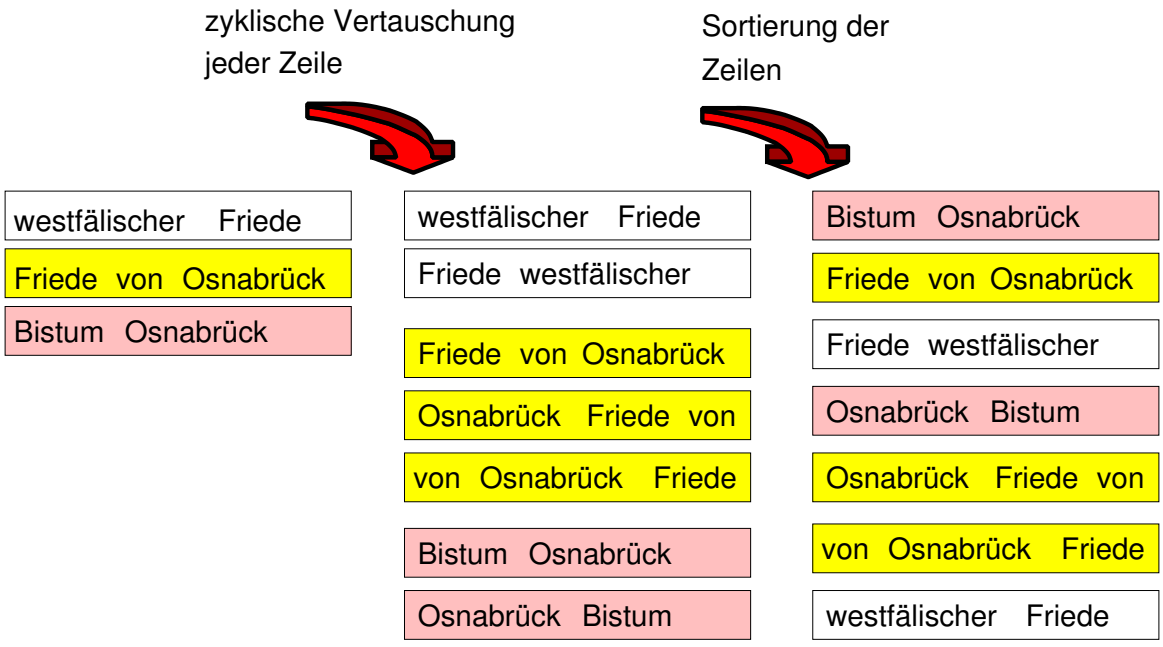
[2] Nein, nicht *der* Michael Jackson.

Bewertung von Modularisierung

Software Architecture Analysis Method (SAAM) (Kazman u. a. 1996):

1. Lege wichtige Qualitätsaspekte fest
2. Beschreibe alternative Modularisierungen
3. Entwickle Szenarien für die Qualitätsaspekte
4. Spiele die Szenarien durch und bewerte die Modularisierungen
5. Betrachte Wechselwirkungen zwischen den Qualitätsaspekten
6. Fasse die Evaluation zusammen

Beispiel: Key Word in Context (KWIC) (Parnas 1972)

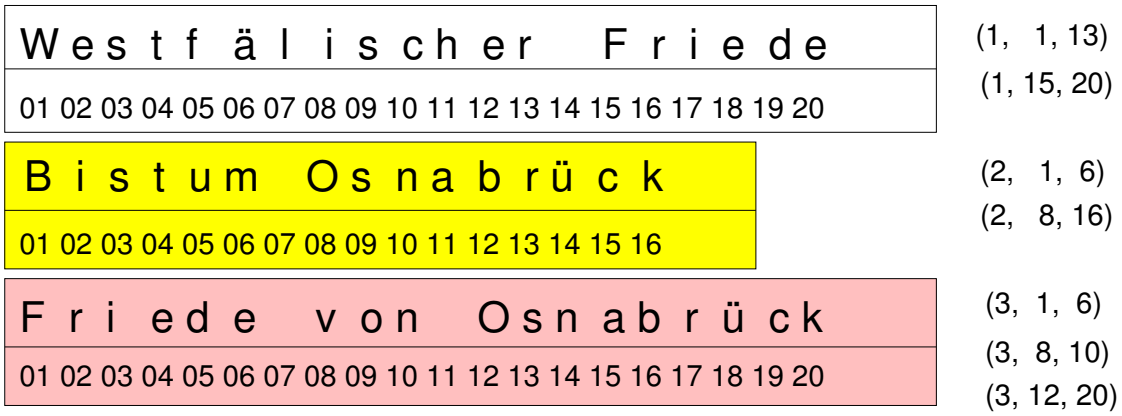


Betrachtete Qualitätsaspekte und Szenarien

- Änderbarkeit
  - Eingabeformat ändert sich
  - größere Dateien müssen verarbeitet werden
  - riesige Dateien müssen verarbeitet werden
- separate Entwickelbarkeit
  - verteile Arbeit an Entwicklungsteams
- Performanz
  - berechne KWIC für FB3-Webseiten

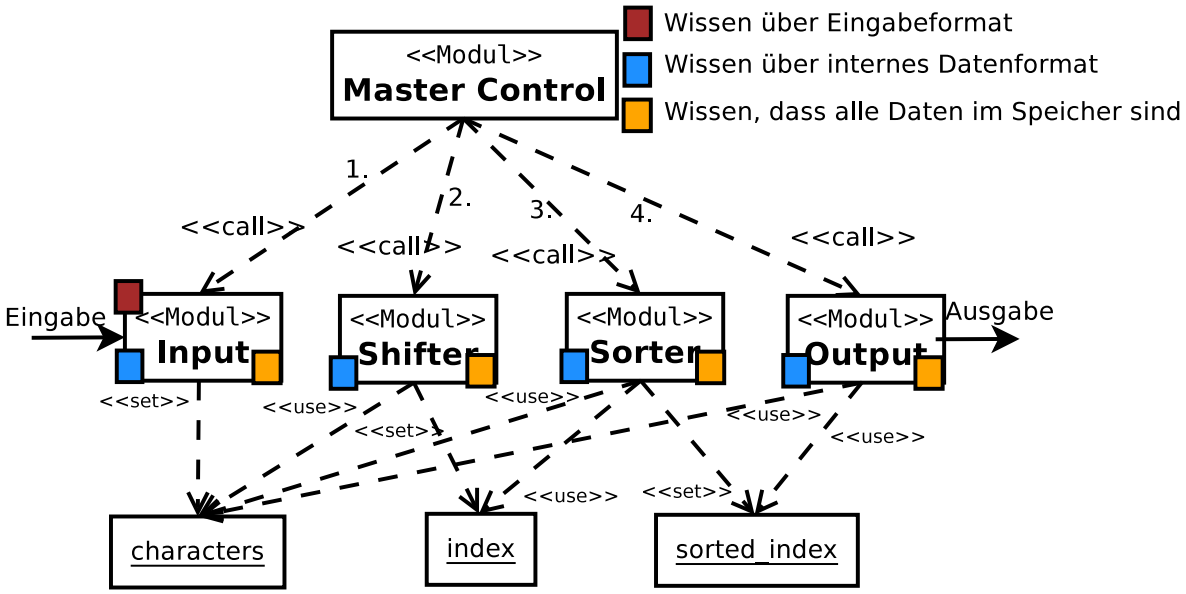
Ablauf

1. Eingabe der Daten
2. Interne Speicherung der Daten
3. Indizierung (Zeile, Wortanfang, Wortende)



4. Zyklische Rotation der Indizierung
5. Sortierung der Indizierung
6. Ausgabe der Indizierung

Erste Modularisierung (ablauforientiert)



Qualitäten

Änderbarkeit:

	betroffene Module			
	Input	Shifter	Sorter	Output
Eingabeformat	×			
größere Dateien	×	×	×	×
riesige Dateien	×	×	×	×

Getrennte Entwicklung:

- alle Datenstrukturen müssen bekannt sein
- komplexe Beschreibung notwendig

Performanz:

- schneller Zugriff auf globale Variablen

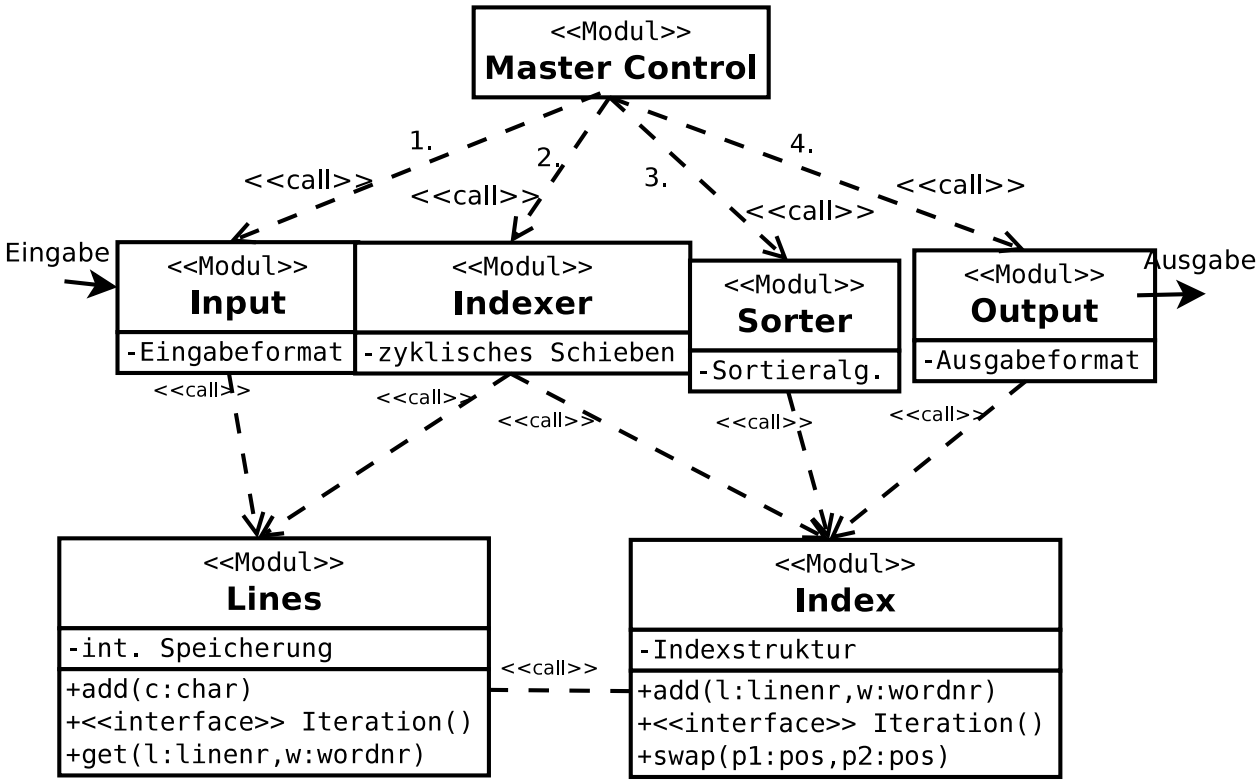
Geheimnisprinzip (Information Hiding) (Parnas 1972)

Definition

Geheimnisprinzip: Module verbergen alle Implementierungsentscheidungen, die sich ändern können, hinter einer abstrakten Schnittstelle.

Zweite Modularisierung (objektbasiert)

. . . nach dem Geheimnisprinzip (Information Hiding) (Parnas 1972)



Qualitäten

Änderbarkeit:

	betroffene Module					
	Input	Lines	Index	Indexer	Sorter	Output
Eingabeformat	×					
größere Dateien		×				
riesige Dateien		×				

Getrennte Entwicklung:

- nur Schnittstellen müssen bekannt sein

Performanz:

- zusätzliche Aufrufe

Abschließendes Wort zur Modularisierung

Es gibt viele Modularisierungsmöglichkeiten.

Jede ist gut für einen Zweck, schlecht für einen anderen.

Mit gängigen Programmiersprachen muss man sich auf eine festlegen.

Definition
Querschnittsbelange (Cross-Cutting Concerns):
Implementierungsaspekte, die eine große Zahl von Modulen betreffen.

Beispiele: Fehlerbehandlung, Logging-Mechanismen, Vermeidung von Speicherlöchern etc.

Aspektorientierte Programmiersprachen erlauben eine separate Beschreibung dieser Aspekte und ein „Einweben“ des Aspekts in das System.

Wiederholungsfragen I

- Was ist Software-Architektur und welche Bedeutung hat sie?
- Welche Faktoren spielen eine Rolle für die Architektur?
- Erläutern Sie den Prozess von Hofmeister et al. zum Entwurf einer Architektur (globale Analyse, Faktoren, Strategien, Blickwinkelentwurf).
- Was ist ein Architekturblickwinkel bzw. eine Architektursicht?
- Erläutern Sie die Blickwinkel von Hofmeister et al. Wer hat jeweils ein Interesse an diesen Blickwinkeln?
- Warum verschiedene Blickwinkel? Warum gerade diese vier Blickwinkel?
- Wie lautet das Gesetz von Conway?
- Was ist Modularisierung?
- Was ist eine Schnittstelle?
- Was ist das Prinzip des Information-Hidings?
- Was versteht man unter Kopplung und Zusammenhalt?
- Nennen Sie Regeln für einen guten Architekturentwurf.

Wiederholungsfragen II

- Wie lässt sich Architektur prüfen? Erläutern Sie insbesondere SAAM.

Weiterführende Literatur

- Bass u. a. (2003) geben eine gute Übersicht zu allen relevanten Aspekten von Software-Architektur
- Hofmeister u. a. (2000) beschreiben eine Methode für den Architekturentwurf, die auf vier verschiedenen Architektursichten beruht
- Shaw und Garlan (1996) geben eine Einführung in Software-Architektur und beschreiben einige Architekturstile bzw. -muster

1 **Bass u. a. 2003** Bass, Len ; Clements, Paul ; Kazman, Rick: Software Architecture in Practice. 2nd edition. Addison Wesley, 2003

2 **Clements u. a. 2002** Clements, Paul ; Bachmann, Felix ; Bass, Len ; Garlan, David ; Ivers, James ; Little, Reed ; Nord, Robert ; Stafford, Judith: Documenting Software Architecture. Boston : Addison-Wesley, 2002

3 **Hofmeister u. a. 2000** Hofmeister, Christine ; Nord, Robert ; Soni, Dilip: Applied Software Architecture. Addison Wesley, 2000 (Object Technology Series)

4 **IEEE P1471 2002** : IEEE Recommended Practice for Architectural Description of Software-intensive Systems—Std. 1471-2000. ISBN 0-7381-2518-0, IEEE, New York, NY, USA. 2002

5 **Kazman u. a. 1996** Kazman, Rick ; Abowd, Gregory ; Bass, Len ; Clements, Paul: Scenario-Based Analysis of Software Architecture. In: IEEE Software (1996), November, S. 47–55

6 Kruchten 1995 Kruchten, Phillipe: The 4+1 View Model of Architecture. In: IEEE Software 12 (1995), November, Nr. 6, S. 42–50

7 Parnas 1972 Parnas, David L.: On the Criteria to Be Used in Decomposing Systems into Modules. In: Communications of the ACM 15 (1972), Dezember, Nr. 12

8 Perry und Wolf 1992 Perry, Dewayne E. ; Wolf, Alexander L.: Foundations for the Study of Software. In: ACM SIGSOFT 17 (1992), Oktober, Nr. 4, S. 40–52

9 Shaw und Garlan 1996 Shaw, Mary ; Garlan, David: Software Architecture – Perspectives on an Emerging Discipline. Upper Saddle River, NJ : Prentice Hall, 1996. – ISBN ISBN 0-13-182957-2

10 Sowa und Zachman 1992 Sowa, J. F. ; Zachman, J. A.: Extending and formalising the framework for information systems architecture. In: IBM Systems Journal (1992)

11 Störrle 2005 Störrle, Harald: UML 2 für Studenten. Pearson Studium, 2005. – ISBN 3-8273-7143-0

12 Zachman 1987 Zachman, J. A.: A framework for information systems architecture. In: IBM Systems Journal 26 (1987), Nr. 3

13 Zachman 1999 Zachman, J. A.: A framework for information systems architecture. In: IBM Systems Journal 38 (1999), Nr. 2&3, S. 454—470